

Digitized by Google

Daniel Danciu George Mardale

ARTA PROGRAMĂRII ÎN JAVA
vol I - Concepte fundamentale

Digitized by Google

Cuprins

Digitized by Google

Digitized by Google

Digitized by Google

Digitized by Google

Digitized by Google

Digitized by Google

Digitized by Google

Digitized by Google

Introducere

"Ce învăţătură predă Maestrul?" - întrebă un vizitator.
"Nici una", răspunse discipolul.
"Atunci de ce ţine discursuri?"
"El doar ne arată calea - nu ne învaţă nimic."

Anthony de Mello, O clipă de înţelepciune

Oricine a folosit cel puţin o dată Internetul sau a citit o revistă de specialitate în domeniul informaticii, a auzit cu siguranţă cuvântul „Java". Java reprezintă un limbaj de programare, creat de compania Sun Microsystems în anul 1995. Iniţial, Java a fost gândit pentru a îmbunătăţi conţinutul paginilor web prin adăugarea unui conţinut dinamic: animaţie, multimedia etc. În momentul lansării sale, Java a revoluţionat Internetul, deoarece era prima tehnologie care oferea un astfel de conţinut. Ulterior au apărut şi alte tehnologii asemănătoare (cum ar fi Microsoft ActiveX sau Macromedia Shockwave[1]), dar Java şi-a păstrat importanţa deosebită pe care a dobândit-o în rândul programatorilor, în primul rând datorită facilităţilor pe care le oferea. Începând cu anul 1998, când a apărut versiunea 2 a limbajului (engl. Java 2 Platform), Java a fost extins, acoperind şi alte direcţii de dezvoltare: programarea aplicaţiilor *enterprise* (aplicaţii de tip server), precum şi a celor adresate dispozitivelor cu resurse limitate, cum ar fi telefoane mobile, *pager*-e sau PDA-uri[2]. Toate acestea au reprezentat facilităţi

[1]Cei care utilizează mai des Internetul sunt probabil obişnuiţi cu controale ActiveX sau cu animaţii flash în cadrul paginilor web.

[2]PDA = Personal Digital Assistant (mici dispozitive de calcul, de dimensiuni puţin mai mari decât ale unui telefon mobil, capabile să ofere facilităţi de agendă, dar şi să ruleze aplicaţii într-un mod relativ asemănător cu cel al unui PC). La momentul actual există mai multe tipuri de PDA-uri: palm-uri, pocketPC-uri, etc.

Digitized by Google

noi adăugate limbajului, care a păstrat însă şi posibilităţile de a crea aplicaţii standard, de tipul aplicaţiilor în linie de comandă sau aplicaţii bazate pe GUI[3]. Lansarea versiunii 2 a limbajului Java a fost o dovadă a succesului imens de care s-au bucurat versiunile anterioare ale limbajului, dar şi a dezvoltării limbajului în sine, a evoluţiei sale ascendente din punct de vedere al facilităţilor pe care le oferă, cât şi al performanţelor pe care le realizează.

Cum este organizată această carte?

Având în vedere popularitatea extraordinară de care se bucură limbajul Java în cadrul programatorilor din întreaga lume, am considerat utilă scrierea unei lucrări în limba română care să fie accesibilă celor care doresc să înveţe sau să aprofundeze acest limbaj. Ideea care a stat la baza realizării acestei cărţi a fost aceea de a prezenta nu numai limbajul Java în sine, ci şi modul în care se implementează algoritmii şi structurile de date fundamentale în Java, elemente care sunt indispensabile oricărui programator cu pretenţii. Aşadar, cartea nu este destinată numai celor care doresc să acumuleze noţiuni despre limbajul Java în sine, ci şi celor care intenţionează să îşi aprofundeze şi rafineze cunoştinţele despre algoritmi şi să îşi dezvolte un stil de programare elegant. Ca o consecinţă, am structurat cartea în două volume: prima volum (cel de faţă) este orientat spre prezentarea principalelor caracteristici ale limbajului Java, în timp ce volumul al doilea (disponibil separat) constituie o abordare a algoritmilor din perspectiva limbajului Java. Finalul primului volum cuprinde un grup de cinci anexe, care prezintă mai amănunţit anumite informaţii cu caracter mai special, deosebit de utile pentru cei care ajung să programeze în Java. Am ales această strategie deoarece a dobândi cunoştinţe despre limbajul Java, fără a avea o imagine clară despre algoritmi, nu reprezintă un progres real pentru un programator. Iar scopul nostru este acela de a vă oferi posibilitatea să deveniţi un programator pentru care limbajul Java şi algoritmii să nu mai constituie o necunoscută.

Cele două volume cuprind numeroase soluţii Java complete ale problemelor prezentate. Mai este necesară o remarcă: deseori am optat, atât în redactarea codului sursă cât şi în prezentarea teoretică a limbajului sau a algoritmilor, pentru păstrarea terminologiei în limba engleză în defavoarea limbii române. Am luat această decizie, ţinând cont de faptul că mulţi termeni s-au consacrat în acest format, iar o eventuală traducere a lor le-ar fi denaturat înţelesul.

Primul volum cuprinde opt capitole:

Capitolul 1 reprezintă o prezentare de ansamblu a tehnologiei Java. Capitolul debutează cu istoria limbajului, începând cu prima versiune şi până la cea

[3]GUI = Graphical User Interface, interfaţă grafică de comunicare cu utilizatorul, cum sunt în general aplicaţiile disponibile pe sistemul de operare Microsoft Windows.

Digitized by Google

curentă. În continuare, sunt înfăţişate câteva detalii despre implementările existente ale limbajului. Implementarea Java a firmei Sun este tratată separat, în detaliu, fiind şi cea pe care s-au testat aplicaţiile realizate pe parcursul cărţii.

Capitolul 2 este cel care dă startul prezentării propriu-zise a limbajului Java, începând cu crearea şi executarea unui program simplu. Sunt prezentate apoi tipurile primitive de date, constantele, declararea şi iniţializarea variabilelor, operatorii de bază, conversiile, instrucţiunile standard şi metodele.

Capitolul 3 este destinat referinţelor şi obiectelor. Sunt prezentate în detaliu noţiunile de referinţă, obiect, şiruri de caractere (String-uri) şi de şiruri de elemente cu dimensiuni variabile.

Capitolul 4 continuă prezentarea începută în capitolul anterior, prezentând modul în care se pot defini propriile clase în Java şi cum se implementează conceptele fundamentale ale programării orientate pe obiecte.

Capitolul 5 prezintă în detaliu un principiu esenţial al programării orientate pe obiecte: *moştenirea*. Sunt prezentate de asemenea noţiuni adiacente cum ar fi cea de polimorfism, interfaţă, clasă interioară, identificare a tipurilor de date în faza de execuţie (RTTI = Runtime Type Identification).

Capitolul 6 este dedicat în întregime modului de tratare a excepţiilor în Java. Se prezintă tipurile de excepţii existente în limbajul Java, cum se pot defini propriile tipuri de excepţii, cum se pot prinde şi trata excepţiile aruncate de o aplicaţie. Finalul capitolului este rezervat unei scurte liste de sugestii referitoare la utilizarea eficientă a excepţiilor.

Capitolul 7 prezintă sistemul de intrare-ieşire (I/O) oferit de limbajul Java. Pe lângă operaţiile standard realizate pe fluxurile de date (*stream*) şi fişiere secvenţiale, este prezentată şi noţiunea de *colecţie de resurse* (engl. resource bundles).

Capitolul 8 este rezervat problemei delicate a firelor de execuţie (*thread*-uri). Pornind cu informaţii simple despre firele de execuţie, se continuă cu accesul concurent la resurse, sincronizare, monitoare, coordonarea firelor de execuţie, cititorul dobândind în final o imagine completă asupra sistemului de lucru pe mai multe fire de execuţie, aşa cum este el atât de elegant realizat în Java.

Cele opt capitole ale primului volum sunt urmate de un grup de anexe, care conţin multe informaţii utile programatorilor Java.

Anexa A constituie o listă cu editoarele şi mediile integrate pentru dezvoltarea aplicaţiilor Java, precum şi un mic tutorial de realizare şi executare a unei aplicaţii Java simple. Tot aici este prezentată şi *ant*, o unealtă de foarte mare ajutor în compilarea şi rularea aplicaţiilor de dimensiuni mai mari.

Anexa B este dedicată convenţiilor de scriere a programelor. Sunt prezen-

Digitized by Google

tate principalele reguli de scriere a unor programe lizibile, conforme cu standardul stabilit de Sun Microsystems. Ultima parte a anexei este dedicată unei unelte foarte utile în documentarea aplicaţiilor Java: *javadoc*.

Anexa C detaliază ideea de pachete, oferind informaţii despre pachete Java predefinite şi despre posibilitatea programatorului de a defini propriile sale pachete. Un accent deosebit se pune şi pe prezentarea arhivelor `jar`.

Anexa D prezintă modul în care se pot realiza aplicaţii internaţionalizate, prin care textele care apar într-o aplicaţie sunt traduse dintr-o limbă în alta, cu un efort minim de implementare. De asemenea, este prezentat şi rolul colecţiilor de resurse (engl. resource bundles) în internaţionalizarea aplicaţiilor.

Anexa E reprezintă o listă de resurse disponibile programatorului Java, pornind de la *site*-ul Sun Microsystems şi până la tutoriale, cărţi, reviste online, liste de discuţii disponibile pe Internet. Cu ajutorul acestora, un programator Java poate să îşi dezvolte aptitudinile de programare şi să acumuleze mai multe cunoştinţe despre limbajul Java.

Volumul al doilea este destinat prezentării algoritmilor. Independenţa algoritmilor relativ la un anumit limbaj de programare, face ca majoritatea programelor din această parte să fie realizate şi în pseudocod, punctând totuşi pentru fiecare în parte specificul implementării în Java.

Capitolul 9 constituie primul capitol al celui de-al doilea volum şi prezintă modalitatea prin care se poate realiza analiza eficienţei unui algoritm. Notaţia asimptotică, tehnicile de analiză a algoritmilor, algoritmii recursivi constituie principala direcţie pe care se axează acest capitol.

Capitolul 10 reprezintă o incursiune în cadrul structurilor de date utilizate cel mai frecvent în conceperea algoritmilor: stive, cozi, liste înlănţuite, arbori binari de căutare, tabele de repartizare şi cozi de prioritate. Fiecare dintre aceste structuri beneficiază de o prezentare în detaliu, însoţită de o implementare Java în care se pune accent pe separarea interfeţei structurii de date de implementarea acesteia.

Capitolul 11 constituie startul unei suite de capitole dedicate metodelor fundamentale de elaborare a algoritmilor. Primul capitol din această suită este rezervat celei mai elementare metode: *backtracking*. După o analiză amănunţită a caracteristicilor acestei metode (cum ar fi cei patru paşi standard în implementarea metodei: *atribuie şi avansează*, *încercare eşuată*, *revenire*, *revenire după construirea unei soluţii*), sunt prezentate câteva exemple de probleme clasice care admit rezolvare prin metoda backtracking: generarea permutărilor, a aranjamentelor şi a combinărilor, problema damelor, problema colorării hărţilor.

Capitolul 12 prezintă o altă metodă de elaborare a algoritmilor: *divide et impera*. Prima parte a capitolului este rezervată prezentării unor noţiuni intro-

Digitized by Google

ductive despre recursivitate şi recurenţă, absolut necesare înţelegerii modului în care funcţionează această metodă. Analog capitolului 11, prezentarea propriu-zisă a metodei este însoţită de câteva exemple de probleme clasice rezolvabile prin această metodă: căutarea binară, sortarea prin interclasare (*mergesort*), sortarea rapidă (*quicksort*), trecerea expresiilor aritmetice în formă poloneză postfixată.

Capitolul 13 prezintă cea de-a treia metodă de elaborare a algoritmilor: *metoda Greedy*. Capitolul păstrează aceeaşi structură ca şi cele precedente: sunt prezentate mai întâi elementele introductive ale metodei, urmate apoi de câteva exemple clasice de probleme rezolvabile prin această metodă: problema spectacolelor, minimizarea timpului de aşteptare, interclasarea optimă a mai multor şiruri ordonate.

Capitolul 14 este rezervat unei metode speciale de elaborare a algoritmilor: *programarea dinamică*, ce reprezintă probabil cea mai complexă metodă de elaborare a algoritmilor, punând deseori în dificultate şi programatorii experimentaţi. Totuşi avem credinţa că modul simplu şi clar în care sunt prezentate noţiunile să spulbere mitul care înconjoară această metodă. Capitolul debutează cu o fundamentare teoretică a principalelor concepte întâlnite în cadrul metodei. Apoi, se continuă cu rezolvarea unor probleme de programare dinamică: înmulţirea unui şir de matrice, subşirul comun de lungime maximă, distanţa Levensthein etc.

Capitolul 15 reprezintă ultimul capitol din seria celor dedicate metodelor de elaborare a algoritmilor. Metoda *branch and bound* este cea abordată în cadrul acestui capitol, prin intermediul unui exemplu clasic de problemă: jocul de *puzzle* cu 15 elemente.

Capitolul 16 reprezintă o sinteză a metodelor de elaborare a algoritmilor care au fost tratate de-a lungul volumului al doilea, prezentând aspecte comune şi diferenţe între metode, precum şi aria de aplicabilitate a fiecărei metode în parte.

Cui se adresează această carte?

Lucrarea de faţă nu se adresează începătorilor, ci persoanelor care stăpânesc deja, chiar şi parţial, un limbaj de programare. Cititorii care au cunoştinţe de programare în C şi o minimă experienţă de programare orientată pe obiecte vor găsi lucrarea ca fiind foarte uşor de parcurs. Nu sunt prezentate noţiuni elementare specifice limbajelor de programare cum ar fi funcţii, parametri, instrucţiuni etc. Nu se presupune cunoaşterea unor elemente legate de programarea orientată pe obiecte, deşi existenţa lor poate facilita înţelegerea noţiu-

Digitized by Google

nilor prezentate. De asemenea, cartea este foarte utilă şi celor care doresc să aprofundeze studiul algoritmilor şi modul în care anumite probleme clasice de programare pot fi implementate în Java.

Pe Internet

Pentru comoditatea cititorilor, am decis să punem la dispoziţia lor codul sursă complet al tuturor programelor prezentate pe parcursul celor două volume ale lucrării în cadrul unei pagini web concepută special pentru interacţiunea cu cititorii. De asemenea, pagina web a cărţii va găzdui un forum unde cititorii vor putea oferi sugestii în vederea îmbunătăţirii conţinutului lucrării, vor putea schimba opinii în legătură cu diversele aspecte prezentate, adresa întrebări autorilor etc. Adresele la care veţi găsi aceste informaţii sunt:

- `http://www.albastra.ro/carti/v178/`
- `http://www.danciu.ro/apj/`

Mulţumiri

În încheiere, dorim să adresăm mulţumiri colegilor şi prietenilor noştri care ne-au acordat ajutorul în realizarea acestei lucrări: Vlad Petric (care a avut o contribuţie esenţială la structurarea capitolului 14), Alexandru Băluţ (autor al anexei A), Iulia Tatomir (a parcurs şi comentat cu multă răbdare de mai multe ori întreaga carte) şi Raul Furnică (a parcurs şi comentat capitolele mai delicate ale lucrării).

Digitized by Google

1. Prezentarea limbajului Java

> Descoperirea constă în a vedea ceea ce toată lumea a văzut şi în a gândi ceea ce nimeni nu a gândit.
>
> Albert Szent-Gyorgi

În cadrul acestui capitol vom prezenta informaţii cu caracter general care să vă permită realizarea unei idei de ansamblu asupra limbajului Java. Sunt prezentate informaţii despre următoarele subiecte:

- De ce este important să învăţaţi Java şi de ce este Java un concurent serios pentru celelalte limbaje de programare;
- Cum a fost creat limbajul Java şi care a fost evoluţia sa în anii care au urmat de la apariţia sa în 1995;
- Care sunt companiile care implementează limbajul Java;
- Prezentare detaliată a celei mai importante implementări a limbajului Java, cea oferită de Sun Microsystems, incluzând informaţii despre terminologia de bază utilizată;
- Ce cuprinde platforma Java oferită de Sun Microsystems.

1.1 Limbaje de programare

Probabil că primul lucru la care se gândesc cei care cunosc şi alte limbaje de programare este la ce le foloseşte asimilarea unui nou limbaj? Nu sunt de ajuns cele pe care le cunosc deja? Oricât de simplu ar părea, un răspuns ferm la această întrebare nu este chiar atât de uşor de formulat. Deşi sunt numeroase

Digitized by Google

limbaje de programare în lumea calculatoarelor, între ele există multe diferenţe. Fiecare are avantaje şi dezavantaje în raport cu celelalte. Nu există un limbaj de programare perfect care să înglobeze toate avantajele oferite de limbajele existente.

Deşi numărul de limbaje de programare este relativ mare, limbajele folosite în dezvoltarea de aplicaţii software sunt în general aceleaşi. Java face parte din această categorie, pentru că este un limbaj de programare remarcabil, care uşurează sarcina programatorilor de a realiza aplicaţii de o complexitate incredibilă. Experienţa în domeniul software a arătat că învăţarea limbajului Java poate să îmbunătăţească nivelul calitativ al unui programator. Pe lângă avantajele practice de a avea mai multe posibilităţi de a rezolva o problemă, învăţarea acestui limbaj poate să lărgească perspectivele şi modalităţile de abordare a problemelor care sunt dificil de implementat în celelalte limbaje. Să considerăm de exemplu cazul în care aveţi deja experienţă de programare în C sau C++. Este foarte dificil să creaţi un program Java fără să înţelegeţi noţiunile de clase şi obiecte, spre deosebire de C++, unde se poate folosi vechiul stil de programare procedural, specific limbajului C. A învăţa Java presupune a învăţa programare orientată pe obiecte (engl. object oriented programming, pe scurt OOP[1]), ceea ce vă poate ajuta cu siguranţă să deveniţi un programator mai bun. Iar Java oferă un înţeles mult mai clar unor concepte mai complexe, cum este OOP, decât reuşesc alte limbaje. De asemenea, în cazul multor probleme, veţi putea scrie mai rapid o soluţie în Java decât în C sau C++. Mai mult, datorită structurii sale, soluţia Java va fi mai puţin predispusă erorilor (vezi problemele spinoase cauzate de operaţiile cu *pointer*i).

Lucrarea de faţă urmăreşte să vă prezinte de-a lungul capitolelor sale într-un mod cât mai atractiv tehnologia Java şi, totodată, să vă adâncească şi mai mult convingerea că acest limbaj se află printre cele care oferă cele mai puternice facilităţi în domeniul software. Iată numai câteva dintre caracteristicile care vor fi studiate în amănunţime: tipuri de date primitive, operatori, metode, referinţe, obiecte, clase, OOP, pachete, moştenire, interfeţe, clase interioare, RTTI (RunTime Type Identification), mecanismul *reflection*, excepţii, operaţii de intrare-ieşire (I/O), fire de execuţie, convenţii de scriere a programelor Java, *javadoc*, colecţii de resurse, aplicaţii internaţionalizate etc.

[1]OOP = modalitate de a conceptualiza un program ca un set de obiecte aflate în interacţiune, având scopul de a îndeplini o sarcină.

Digitized by Google

1.2 Începuturile limbajului Java

Spre deosebire de alte limbaje de programare, este foarte probabil ca informaţiile pe care le aveţi despre tehnologia Java să nu fie mai vechi de câţiva ani. Motivul este simplu: Java a apărut doar cu câţiva ani în urmă, în 1995. 23 mai 1995 a fost ziua în care a fost lansată oficial tehnologia care a schimbat radical lumea calculatoarelor în cei aproape opt ani de la apariţie. În acea zi, John Gage, directorul departamentului "Science Office" din cadrul Sun Microsystems, şi Marc Andreessen, co-fondator şi vicepreşedinte al firmei Netscape, au anunţat în cadrul conferinţei SunWorld că tehnologia Java a devenit realitate şi că urmează să fie încorporată în Netscape Navigator, browser-ul pentru Internet al firmei Netscape. La acea dată, întreaga echipă de la Sun care concepuse Java avea mai puţin de 30 de membri.

Începuturile limbajului Java datează însă dinainte de 1995. Tehnologia Java a fost creată ca o unealtă de programare în cadrul unui proiect privat de dimensiuni reduse, iniţiat în cadrul companiei Sun, în anul 1991, de către un mic grup care îi cuprindea printre alţii pe Patrick Naughton, Mike Sheridan şi James Gosling. Proiectul purta numele "The Green Project" şi nu avea ca scop crearea unui nou limbaj de programare. Grupul, format din 13 membri, primise de la Sun Microsystems sarcina de a anticipa şi a plănui "noul val" în lumea tehnologiei informaţiei (engl. IT). După o perioadă de timp în care au analizat factorii implicaţi, concluzia membrilor grupului a fost că o tendinţă importantă în următorii ani urma să fie dezvoltarea dispozitivelor electrocasnice "inteligente" - televizoare interactive, iluminat interactiv, etc. Cercetătorii Sun considerau că viitorul era rezervat dispozitivelor care puteau comunica între ele, în aşa fel încât, de exemplu, televizorul să-i comunice telefonului mobil că a început emisiunea dumneavoastră preferată, iar acesta din urmă să vă alerteze printr-un semnal acustic. Pentru a demonstra ce văd ei ca fiind viitorul dispozitivelor digitale, cei 13 membri au prezentat în vara lui 1992, după 18 luni de muncă neîntreruptă, un dispozitiv interactiv cu interfaţă grafică, asemănător unui mic televizor portabil. În acea demonstraţie, de acum cunoscuta mascotă a tehnologiei Java, ducele ("The Duke"), executa câteva animaţii pe ecran. Prototipul era denumit *7 ("StarSeven") şi se remarca prin faptul că era capabil să comunice şi cu alte dispozitive. *7 funcţiona cu ajutorul unui limbaj cu totul nou. Iniţial, ideea a fost de a realiza sistemul de operare al *7 în C++, ultrapopularul limbaj de programare orientat pe obiecte, dar până la urmă s-a decis crearea unui nou limbaj de programare, special pentru *7, care să comunice mai bine cu el. Limbajul a fost creat de James Gosling, membru al grupului, care l-a denumit "Oak" ("stejar"), după copacul din faţa ferestrei sale. Mai târziu, Sun a descoperit că acest nume era deja folosit pentru un alt produs şi l-a redenu-

Digitized by Google

mit în *Java*. Numele a fost ales după multe ore de gândire, dintr-o listă cu numeroase alte alternative. Ideea membrilor grupului era să găsească un nume care să cuprindă esenţa tehnologiei: vioiciune, animaţie, viteză, interactivitate etc. Java nu reprezenta un acronim, ci mai degrabă o amintire a acelui lucru fierbinte şi aromat pe care mulţi programatori îl beau în mari cantităţi (pentru cei care nu s-au prins, este vorba de cafea).

Fiind proiectat pentru a fi folosit de dispozitive electrocasnice şi nu de calculatoare super-performante, Java trebuia să fie mic, eficient şi să funcţioneze pe o gamă largă de dispozitive hardware. De asemenea, trebuia să fie fiabil. Utilizatorii de calculatoare s-au mai obişnuit cu clasicele "căderi" ale sistemului, însă această situaţie nu putea fi acceptabilă pentru utilizatorii de aparatură electrocasnică (majoritatea utilizatorilor de aparatură electrocasnică au dificultăţi în a învăţa cum să manipuleze corect noul aparat - cum ar fi dacă acesta ar mai şi reacţiona uneori complet neprevăzut).

Pe măsură ce în cadrul proiectului s-au implicat potenţiali clienţi, acesta a devenit public, iar grupul şi-a schimbat numele din "The Green Team" în "FirstPerson". Cu timpul, echipa "FirstPerson" a început să caute o piaţă de desfacere pentru dispozitivul *7, iar industria TV părea să fie cea mai potrivită pentru acest lucru. Echipa a creat un nou demo, denumit "MovieWood", pentru a demonstra potenţialul tehnologiei pe piaţa dispozitivelor TV. Din păcate, industria televizoarelor digitale se afla în perioada de început şi proiectul nu a mai înaintat în această direcţie. Următorul pas a fost încercarea de a convinge companiile de televiziune prin cablu să utilizeze noua tehnologie. Utilizarea ei ar fi transformat aceste companii în adevărate reţele interactive, asemănătoare Internet-ului din zilele noastre, reţele în care utilizatorii ar fi putut citi sau scrie informaţii, dar companiile de cablu au considerat că ar putea pierde controlul asupra reţelelor, tentativa de promovare fiind şi de această dată sortită eşecului.

Curând ei au realizat că noua tehnologie nu poate pătrunde în industria TV şi s-au decis să încerce Internetul, care la acea vreme de abia începuse să câştige în popularitate. Internetul reprezenta exact tipul de reţea pe care echipa îl imaginase pentru industria TV şi a televiziunilor prin cablu. Internetul devenise un mijloc popular de a transporta informaţii multimedia (text, imagini, filme) în cadrul unei reţele şi se asemăna din acest punct de vedere cu tehnologia Java. Deşi Internetul avea deja 20 de ani de când apăruse, fusese dificil de folosit până la apariţia programului *Mosaic* în 1993, pentru că permitea doar folosirea tehnologiilor *ftp* (FileTransferProtocol) sau *telnet*. Apariţia *Mosaic* a revoluţionat modul în care oamenii percepeau Internetul, deoarece avea o interfaţă grafică foarte uşor de utilizat. A fost momentul în care membrii grupului au ajuns la concluzia că Java trebuie să devină o tehnologie destinată Internetului.

Primul pas făcut de grup în această direcţie a fost crearea unei clone a *Mo-*

Digitized by Google

saic, bazată pe tehnologie Java. Aplicaţia a fost denumită *WebRunner* (după filmul "Blade Runner"), dar a devenit oficial cunoscută sub numele de *HotJava*. Era 1994, anul apariţiei primului *browser* bazat pe tehnologie Java. Deşi era la stadiul de aplicaţie demo, *HotJava* era impresionant: pentru prima dată un browser de Internet oferea conţinut animat. La începutul anului 1995, Gosling şi Gage au făcut o primă prezentare publică a noului browser *HotJava*, care a avut un succes răsunător. Astfel că la începutul lunii martie 1995, versiunea alfa (denumită "1.0a2") a produsului *HotJava* a devenit publică pe Internet. Codul sursă al aplicaţiei a fost făcut şi el public. În săptămânile care au urmat, numărul de download-uri al aplicaţiei *HotJava* a crescut impresionant, atingând 10.000 de copii. A fost un număr care a depăşit cu mult până şi aşteptările grupului. Trăsăturile care au făcut Java să fie bun pentru *7 s-au dovedit a fi bune şi pentru Internet. Java era:

- mic - programe de dimensiuni reduse ofereau multe facilităţi;
- portabil - putea fi rulat pe diferite platforme (Windows, Solaris, Linux, etc.) fără modificări;
- sigur - împiedica scrierea de programe care produceau pagube calculatoarelor.

Curând, Sun a realizat că popularitatea tehnologiei Java şi a grupului care o crease, a atins un nivel foarte ridicat şi a trecut la fapte.

Pe 23 mai 1995, tehnologia Java a fost lansată oficial de Sun Microsystems. Evenimentele nu s-au oprit însă aici. În aceeaşi zi, Eric Schmidt şi George Paolini din cadrul Sun Microsystems împreună cu Marc Andreessen din cadrul Netscape au semnat înţelegerea conform căreia tehnologia Java urma să fie integrată în omniprezentul şi omnipotentul (la acea vreme...) browser Netscape Navigator. Surpriza era cu atât mai mare cu cât nici membrii echipei "FirstPerson" nu ştiau nimic despre tratativele dintre cele două părţi.

Cu toate că Java şi *browser*-ul HotJava s-au bucurat de o mare atenţie din partea comunităţii Web, limbajul s-a lansat cu adevărat abia după ce Netscape a fost prima companie care l-a licenţiat. Imediat după prima lansare, Sun şi-a îndreptat eforturile de dezvoltare Java printr-o nouă sucursală: JavaSoft. De-a lungul anilor care au urmat, tehnologia Java a câştigat foarte mult în maturitate, iar potenţialul ei este încă enorm. Au apărut JDK-ul, applet-urile, mii de cărţi despre tehnologia Java, arhitectura JavaBeans, Netscape Communicator, servlet-urile, Java Server Pages (JSP), Java Foundation Classes (JFC), componentele Enterprise JavaBeans (EJB), comerţul electronic, precum şi implicarea în dezvoltarea tehnologiei Java a unor firme deosebit de importante, cum ar fi

Digitized by Google

IBM. Toate aceste lucruri arată cât de rapid s-a dezvoltat tehnologia Java în anii care au trecut de la lansarea ei oficială. Java este acum un mediu important de dezvoltare a aplicaţiilor. Foarte mulţi utilizatori au fost atraşi de obiectivul pe care această tehnologie îl are: *"write once, run anywhere"* ("scris o dată, rulat oriunde"), ceea ce tradus înseamnă că folosind tehnologia Java, acelaşi program devine portabil şi poate fi executat pe mai multe sisteme de operare/platforme (de exemplu: Windows, Linux, Mac, Solaris etc.). De la apariţia ei în mai 1995, platforma Java a fost adoptată în industria IT mai rapid decât orice altă nouă tehnologie din istoria calculatoarelor. Se poate spune că Java a devenit limbajul, platforma şi arhitectura calcului în reţea.

1.3 Caracteristici ale limbajului Java

Limbajul Java se bucură de o imensă popularitate în rândul programatorilor. Puterea sa constă în faptul că este:

- independent de platformă;
- orientat pe obiecte;
- uşor de învăţat şi de folosit în dezvoltarea de aplicaţii software;
- disponibil gratuit, factor care a dus la rapida sa răspândire.

Posibilitatea ca aceeaşi aplicaţie să ruleze nemodificată pe diferite platforme (sisteme de operare) este unul dintre cele mai semnificative avantaje pe care limbajul Java le are asupra altor limbaje de programare. După cum se ştie, atunci când se compilează un program C (sau în alt limbaj de programare), compilatorul transformă fişierul sursă[2] în *cod-maşină* - instrucţiuni specifice sistemului de operare şi procesorului pe care îl foloseşte calculatorul respectiv. Dacă se compilează codul sursă pe un sistem Windows, programul va rula pe un sistem Windows, dar nu va putea rula pe un sistem Linux. Pentru a-l rula pe Linux, codul sursă trebuie transferat pe Linux şi recompilat, pentru a produce cod executabil pe noul sistem. În concluzie, pentru fiecare sistem trebuie produs câte un program executabil separat. Programele Java dobândesc independenţă din punct de vedere al sistemului de operare prin folosirea unei *maşini virtuale*[3] - un fel de calculator în alt calculator. Maşina virtuală preia programul

[2]denumit şi *cod sursă*, reprezintă un set de instrucţiuni pe care un programator le introduce cu ajutorul unui editor de texte, atunci când crează un program

[3]denumită JVM, sau interpretor Java

Digitized by Google

Java compilat şi transformă instrucţiunile în comenzi inteligibile sistemului respectiv. Acest program compilat (într-un format denumit *bytecode*[4]) poate rula pe orice platformă (sistem de operare) care posedă o maşină virtuală Java. Şi nu este vorba numai de PC-uri, ci şi de alte dispozitive, cum ar fi: telefoane mobile, PDA-uri etc. Astfel, programatorii Java nu sunt nevoiţi să creeze versiuni diferite ale programelor pentru fiecare platformă pe care folosesc aplicaţia, deoarece maşina virtuală este cea care realizează "traducerea" necesară. Maşina virtuală nu adaugă un nivel nedorit între sursă şi codul-maşină, fiind indispensabilă pentru a asigura independenţa de platformă.

Orientarea pe obiecte a limbajului Java este o caracteristică logică, ţinând cont de faptul că Java îşi are rădăcinile în limbajul C++, moştenind de la acesta majoritatea conceptelor OOP.

Java a fost modelat foarte asemănător cu C++, majoritatea sintaxei şi a structurii obiect orientate provenind direct din acest limbaj. Pentru un programator C++ este foarte uşor să înveţe Java. În ciuda asemănării cu C++, noţiunile cele mai complexe (şi totodată cele mai mari generatoare de erori) au fost eliminate în Java: pointeri, aritmetică cu pointeri, gestionarea memoriei se face automat (şi nu de către programator). Un program Java este uşor de scris, compilat şi depanat.

1.4 Implementări ale limbajului Java

Tehnologia Java poate fi împărţită în patru componente:

- Maşina virtuală, denumită *JVM* (Java Virtual Machine), interpretează fişierele cu extensia *.class* care conţin bytecode. Bytecode-ul reprezintă cod compilat care este procesat de un program, denumit *maşină virtuală*, spre deosebire de codul executabil normal care rulează pe maşina *reală* care dispune de un procesor hardware. Bytecode-ul este format din instrucţiuni generice care vor fi apoi convertite de către maşina virtuală în instrucţiuni specifice procesorului. Acesta este motivul pentru care aplicaţiile Java sunt independente de platformă. Aplicaţiile Java nu trebuie modificate pentru a le putea rula pe platforme diferite de cea pe care au fost create. Este sarcina maşinii virtuale de pe respectiva platformă să creeze cod executabil specific platformei. Cu alte cuvinte, o aplicaţie

[4]codul maşină pentru maşina virtuală, adică instrucţiuni pe care maşina virtuală le înţelege direct. Codul sursă este compilat în *bytecode*, astfel încât poate fi rulat pe o maşină virtuală Java.

Digitized by Google

Java scrisă sub Windows poate fi rulată şi pe Linux, fără a fi necesară nici măcar o recompilare. Interpretarea byte-code-ului rămâne în sarcina maşinilor virtuale de pe cele două platforme, în cazul nostru Windows şi Linux. Pentru a putea rula aplicaţii Java, transformate prin compilare în fişiere *.class*, este absolut necesară o maşină virtuală. Lipsa acesteia duce la imposibilitatea execuţiei aplicaţiilor Java;

- Limbajul Java propriu-zis. Limbajul Java este orientat pe obiecte şi se aseamănă din punct de vedere al sintaxei cu C++, simplificat însă pentru a elimina elementele de limbaj cauzatoare de erori de programare;

- Un compilator care generează fişierele cu extensia *.class*. Sarcina compilatorului este de a crea, pornind de la programul scris în limbajul Java (adică, de la fişiere cu extensia *.java*), fişierele *.class*, care vor fi interpretate de maşina virtuală;

- Biblioteca de clase Java, denumită şi Java *API* (*engl.* Application Programming Interface). Tehnologia Java include un set de componente puternice şi utile, care pot fi reutilizate de programatori în aplicaţii de natură foarte diversă.

Terminologia Java este uneori destul de confuză. Multe persoane, inclusiv cei de la Sun, folosesc termenul *"Java"* pentru fiecare dintre aceste componente, dar şi pentru tehnologia în ansamblu. Din acest motiv, înţelesul termenilor trebuie adeseori desprins din context. De exemplu, atunci când cineva spune că rulează un program Java, prin aceasta trebuie să înţelegem că de fapt rulează un set de fişiere *.class* pe maşina virtuală *JVM*.

Orice implementare a limbajului Java presupune implicit implementarea tuturor celor patru componente prezentate anterior. Este şi cazul celor mai importante implementări ale limbajului Java, realizate de două nume mari în lumea calculatoarelor: Sun şi IBM. Implementările oferite de cele două mari firme nu sunt singurele. De o importanţă mai redusă sunt: Blackdown JDK (disponibil pe Linux, detalii la adresa `http://www.blackdown.org`), Kaffe (disponibil pe Linux, distribuit cu Linux-ul în sine, detalii la adresa: `http://www.kaffe.org`).

Nici Microsoft nu a rămas datoare la acest capitol şi a creat propria sa implementare Java, disponibilă (evident) numai pe platformele Windows. Implementarea poartă numele de *Microsoft SDK for Java* şi cuprinde un compilator Java şi biblioteca de clase, maşina virtuală fiind download-abilă separat. Merită semnalat aici, ca o paranteză, faptul ca această implementare a limbajului Java a fost obiectul unui proces între Microsoft şi Sun. După trei ani procesul

Digitized by Google

a luat sfârşit, în ianuarie 2001. Sentinţa nu a fost o surpriză pentru lumea IT, pentru că şansele ca Microsoft să câştige erau reduse. Motivul procesului a fost faptul că Microsoft a încălcat în mod intenţionat contractul cu Sun, plasând în mod repetat logo-ul Java deţinut de Sun pe aplicaţii software care nu satisfăceau standardele de compatibilitate impuse de Sun prin contractul dintre cele două părţi. Există păreri care susţin că Microsoft nici nu a dorit să satisfacă standardul de compatibilitate, pentru că astfel produsele sale ar fi devenit disponibile şi pe platformele non-Microsoft. Ca rezultat al sentinţei judecătoreşti, Microsoft a plătit 20 de milioane de dolari companiei Sun, nu va mai primi licenţă Java, nu va mai putea folosi logo-ul Java şi va putea să distribuie doar versiunea 1.1.14 a implementării Java, dar şi aceasta sub o licenţă limitată, care va expira peste şapte ani. Ca răspuns, Microsoft încearcă să convingă dezvoltatorii de aplicaţii software să folosească platforma proprietară ".NET", împreună cu propriul limbaj de programare C#. Rămâne de văzut însă ce succes va avea această platformă.

Dintre implementările Java, cel mai frecvent utilizată este cea realizată de Sun, acesta fiind şi motivul pentru care o vom descrie mai detaliat în paragraful 1.5. Există situaţii în care programatorii optează pentru implementarea IBM, în special în cazul în care Sun nu oferă o implementare Java pentru platforma dorită. Din acest motiv, se cuvine să prezentăm pe scurt şi implementarea celor de la IBM.

IBM oferă implementări ale limbajului Java pentru următoarele platforme (în paranteze este trecută versiunea la care a ajuns implementarea):

- AIX (versiunea 1.3) ;
- Linux (versiunea 1.3);
- AS/400;
- OS/2 (versiunea 1.3);
- OS/390 (versiunea 1.1.8);
- VM/ESA (versiunea 1.1.6);
- Windows (versiunea 1.1.8).

Cu excepţia implementărilor pentru Windows şi Linux, celelalte implementări ale IBM nu sunt disponibile la Sun (în plus faţă de IBM, Sun realizează şi o implementare Java pentru Solaris), aşa că dacă platforma pe care o folosiţi este una dintre acestea, va trebui să apelaţi la implementarea IBM corespunzătoare.

Digitized by Google

Un alt avantaj al implementărilor IBM este compilatorul *jikes*, compilatorul Java realizat de IBM. Acesta este mai rapid şi mai "prietenos" cu utilizatorul, decât compilatorul oferit de Sun.

Implementările IBM trec printr-un proces complex de validare de la realizare şi până la lansarea către publicul larg. Pentru a fi făcute publice, ele trebuie să fie stabile şi să treacă testul de compatibilitate al firmei Sun.

1.5 Implementarea Sun a limbajului Java

Platforma Java se bazează pe arhitectura de reţea şi pe ideea că aceeaşi aplicaţie software trebuie să ruleze pe diferite tipuri de calculatoare, dar şi pe alte dispozitive (de exemplu, sistemul de navigare al unui automobil). De la lansarea ei comercială, în mai 1995, tehnologia Java a crescut în popularitate tocmai datorită portabilităţii ei. Platforma Java permite executarea aceleiaşi aplicaţii Java pe diferite tipuri de calculatoare. Orice aplicaţie Java poate fi uşor transmisă prin Internet sau prin orice tip de reţea, fără probleme de compatibilitate între sisteme de operare sau hardware. De exemplu, se poate rula o aplicaţie bazată pe tehnologie Java pe un PC, pe un calculator Macintosh, sau chiar pe anumite tipuri de telefoane mobile.

Tehnologia Java permite programatorilor şi utilizatorilor să realizeze lucruri care înainte nu erau posibile. Prin intermediul Java, Internetul şi reţelele de calculatoare devin mediul natural de dezvoltare al aplicaţiilor. De exemplu, utilizatorii pot accesa în deplină securitate informaţii personale sau aplicaţii, utilizând un calculator conectat la Internet. De asemenea, ei pot accesa aplicaţii şi de pe un telefon mobil bazat pe tehnologie Java sau pot folosi carduri *Smart*, de dimensiunea unei cărţi de credit, ca modalitate de identificare.

1.5.1 Platformele Java

Datorită faptului că tehnologia Java a evoluat în mai multe direcţii simultan, Sun a grupat tehnologia Java în trei ediţii:

- Java 2 Platform, Micro Edition (platforma J2ME);
- Java 2 Platform, Standard Edition (platforma J2SE);
- Java 2 Platform, Enterprise Edition (platforma J2EE).

Fiecare dintre cele trei ediţii oferă uneltele necesare creării de aplicaţii software: o maşină virtuală Java adaptată resurselor fizice (memorie, procesor, etc.) ale

Digitized by Google

dispozitivului pe care rulează, aplicaţii utilitare pentru compilare şi executare, şi o bibliotecă de clase (API-ul) specializate pentru fiecare tip de dispozitiv.

Ediţiile se adresează programatorilor care doresc să realizeze un anumit tip de aplicaţie software:

- *J2SE* reprezintă mediul de dezvoltare al aplicaţiilor de tip *client-side* (care rulează pe maşina client, cum ar fi appleturile). J2SE este o unealtă de bază pentru crearea de aplicaţii sofisticate şi puternice, datorită uneltelor de dezvoltare şi ierarhiei de clase (*API*) pe care o oferă;

- *J2EE* reprezintă mediul de dezvoltare al aplicaţiilor de tipul *server-side* (care rulează pe un server si pot fi accesate de mii de clienţi simultan, cum ar fi servlet-urile sau Enterprise Java Beans). Tehnologia J2EE uşurează enorm crearea de aplicaţii internet industriale scalabile prin utilizarea de componente *Enterprise JavaBeans* standardizate, modulare şi reutilizabile, şi automatizând multe detalii de comportament ale aplicaţiei, oferind astfel programatorilor mai mult timp pentru a dezvolta partea logică a aplicaţiei fără să piardă timp cu partea de infrastructură;

- *J2ME* se adresează programatorilor care doresc să dezvolte aplicaţii pentru dispozitive cu resurse mai reduse decât un calculator PC obişnuit. Printre aceste dispozitive putem enumera: carduri *Smart*, *pager*-e, telefoane mobile, *PDA*-uri (*engl.* Personal Digital Assistant), anumite tipuri de televizoare.

Având în vedere faptul că fiecare dintre aceste trei ediţii presupune o imensă cantitate de informaţii de acumulat, lucrarea de faţă va prezenta doar tehnologia J2SE.

1.5.2 Platforma J2SE (Java 2 Platform, Standard Edition)

În 1995, când tehnologia Java a fost oficial lansată, produsul Sun care permitea crearea şi executarea programelor Java se numea *JDK* (Java Development Kit). În decembrie 1998, Sun a decis să împartă tehnologia Java în trei direcţii şi a fost necesară introducerea unor nume noi pentru tehnologiile nou create. Aşa s-a ajuns la denumirea *Java 2*. Prin urmare, platforma JDK a fost redenumită *J2SE* (Java 2 Platform, Standard Edition). Celelalte două platforme, J2EE si J2ME, fiind nou apărute, nu a fost necesară redenumirea lor. La momentul apariţiei noii denumiri, produsul JDK era la versiunea 1.2, astfel că el a fost redenumit la rândul său *Java 2 SDK, Standard Edition, v 1.2* (SDK = Software Development Kit).

Digitized by Google

Probabil că aceste denumiri au creat deja o anumită stare de confuzie. În primul rând trebuie făcută diferenţa dintre *platformă* şi *produs*. Prin *platformă* se înţelege o funcţionalitate abstractă, iar prin *produs* se înţelege aplicaţia software care implementează platforma respectivă. Până în decembrie 1998, platforma Java se numea *JDK*, iar produsele se numeau *JDK 1.0*, *JDK 1.1*, în funcţie de versiunea la care ajunsese produsul. În decembrie 1998, sau altfel spus, începând cu versiunea 1.2 a produsului *JDK*, platforma a fost denumită *Java 2 Platform, Standard Edition, v 1.2 (J2SE)*, iar produsul a fost denumit *Java 2 SDK, Standard Edition, v 1.2 (J2SDK)*. Noua denumire se aplică doar produselor JDK de la versiunea 1.2. Versiunile anterioare (JDK 1.0, JDK 1.1.x) îşi păstrează denumirea iniţială. Un alt lucru care trebuie remarcat este că denumirea *Java 2* nu afectează denumirea maşinii virtuale, care este tot JVM, sau a limbajului în sine, care este tot Java.

Aşadar, J2SDK este produsul Sun care implementează platforma abstractă J2SE şi care este folosit pentru dezvoltarea de aplicaţii specifice platformei J2SE. Produsul include uneltele de creare a programelor Java (compilator etc.) şi uneltele de execuţie ale acestora (interpretor, maşina virtuală etc.). Componentele necesare pentru a executa programele Java sunt grupate sub forma unui subprodus, denumit *Java 2 Runtime Environment, Standard Edition (J2RE)*. J2RE reprezintă mediul în care se execută aplicaţiile scrise pentru platforma J2SE. El este un produs Sun de sine stătător şi poate fi folosit şi fără J2SDK, dar în această situaţie nu se vor putea crea aplicaţii Java, ci doar rula unele deja create. Deci, dacă doriţi doar să executaţi aplicaţii Java, nu este obligatoriu să instalaţi pe calculatorul dumneavoastră produsul J2SDK, fiind de ajuns să aveţi J2RE. În schimb, dacă doriţi să creaţi propriile aplicaţii Java, este absolut necesar să aveţi J2SDK, pentru că J2SDK este produsul care cuprinde uneltele de *dezvoltare* a aplicaţiilor Java.

Pentru a vă crea o imagine şi mai bună asupra noţiunii de J2SE, vom prezenta în continuare conţinutul acestei platforme, folosind ca exemplu versiunea curentă a platformei, la momentul redactării acestui paragraf (aprilie 2002). Pentru a fi la curent cu ultimele versiuni ale tuturor platformelor Java oferite de Sun, puteţi afla detalii la adresa de Internet `http://java.sun.com`.

Versiunea curentă a platformei J2SE este 1.4 şi a fost lansată în primăvara anului 2002. Java 2 Platform, Standard Edition v 1.4, denumirea completă a ultimei versiuni a platformei J2SE, este formată din:

- Java 2 SDK, Standard Edition, v 1.4 (SDK);
- Java 2 Runtime Environment, Standard Edition, v 1.4 (JRE);
- Java 2 Platform, Standard Edition, v 1.4 Documentation (DOCS).

Digitized by Google

În continuare, vom prezenta în detaliu fiecare dintre cele trei componente ale J2SE.

1.5.3 J2SDK (Java 2 SDK, Standard Edition)

J2SDK, versiunea 1.4, este disponibilă pentru următoarele platforme (sisteme de operare): Windows, Linux şi Solaris. Sun nu oferă implementări ale limbajului Java pentru alte sisteme de operare. Prin urmare, dacă sistemul de operare pe care îl folosiţi diferă de cele trei enumerate mai sus, va trebui să găsiţi o altă companie care să ofere o implementare a limbajului Java pentru platforma dumneavoastră.

Java 2 SDK reprezintă mediul de dezvoltare pentru aplicaţii, applet-uri şi componente la standarde profesionale, folosind limbajul de programare Java. Java 2 SDK include unelte utile pentru dezvoltarea şi testarea programelor scrise în limbajul de programare Java şi executate pe platforma Java. Aceste unelte sunt proiectate pentru a fi folosite în linie de comandă (de exemplu, promptul MS–DOS pe Windows), cu alte cuvinte, nu au interfaţă grafică. Deşi acest mod de operare pare într-un fel de modă veche, el reprezintă totuşi o metodă foarte eficientă pentru utilizatorii obişnuiţi cu aceste aplicaţii utilitare. Cei mai mulţi utilizatori de PC-uri asociază utilizarea liniei de comandă cu sistemul de operare MS-DOS. Totuşi, uneltele oferite de J2SDK nu sunt aplicaţii create pentru sistemul de operare MS-DOS. De altfel, nici nu există vreo versiune de J2SDK/JDK destinată sistemului MS-DOS. Decizia de utilizare a aplicaţiilor utilitare în linie de comandă a aparţinut dezvoltatorilor de programe Java. Pe de altă parte, faptul că sunt folosite aplicaţii în linie de comandă pentru a compila programele Java, nu înseamnă că nu se pot crea aplicaţii cu interfaţă grafică. Din contră, mediul de dezvoltare Java oferă facilităţi sofisticate pentru interfaţa grafică, de exemplu applet-urile, Swing, AWT (Abstract Window Toolkit) sau JFC. Aceste componente nu vor face însă subiectul lucrării de faţă. Mai mult, există nenumărate editoare şi medii de programare pentru Java, care automatizează procesul compilării. O prezentare a celor mai des utilizate o veţi găsi în cadrul anexei A.

După cum spuneam, produsul J2SDK oferă o multitudine de aplicaţii utilitare pentru compilarea, executarea, verificarea şi managementul programelor Java. Dintre acestea, trei sunt cele mai importante:

- `javac`, compilatorul Java. Este folosit pentru a transforma programele Java într-o formă care poate fi executată;
- `java`, interpretorul Java. Este folosit pentru a rula programele Java compilate cu `javac`;

Digitized by Google

- `appletviewer`, folosit la testarea applet-urilor. Este folosit pentru a executa applet-urile fără a folosi un browser de Internet. Această aplicaţie este singura dintre aplicaţiile utilitare care oferă interfaţă grafică, deşi se lansează în linie de comandă ca şi celelalte unelte.

Pe lângă aceste aplicaţii utilitare, J2SDK mai cuprinde:

- maşina virtuală JVM;
- biblioteca de clase (API-ul);
- documentaţia J2SDK (disponibilă separat).

Documentaţia J2SDK este de o importanţă covârşitoare pentru orice programator, de aceea trebuie să insistăm mai mult asupra acestui subiect. Documentaţia poate fi accesată *online* pe site-ul firmei Sun, dar poate fi şi descărcată pentru a fi utilizată în cazul în care nu există o conexiune Internet disponibilă permanent (cu alte cuvinte, pentru utilizare *offline*). Sun oferă o gamă largă de documentaţii: specificaţiile API, ghidul programatorului, tutoriale, cărţi, etc. Cele mai multe dintre aceste documente sunt în format `HTML` (HyperTextMarkupLanguage), `ps` (PostScript) sau `pdf` (PortableDataFormat).

Specificaţia API prezintă toate clasele din biblioteca de clase pe care J2SDK o oferă. Tot acolo este descrisă funcţionalitatea fiecărei clase şi sunt oferite exemple de utilizare în anumite contexte. Practic, această specificaţie este documentul cel mai utilizat de un programator Java.

Suplimentar, cei interesaţi mai pot găsi:

- aplicaţii demonstrative împreună cu codul sursă, disponibile în directorul `demo` al distribuţiei produsului J2SDK;
- tutorialul Java ("The Java Tutorial") realizat de Sun, care cuprinde sute de exemple complete şi poate fi descărcat de la adresa `http://java.sun.com/docs/books/tutorial/information/download.html`;
- o listă de cărţi utile, disponibilă la adresa `http://java.sun.com/docs/books`.

Informaţii complete despre documentaţia J2SDK, v 1.4 se pot afla la adresa de Internet: `http://java.sun.com/j2se/1.4/docs/index.html`.

Pentru a obţine produsul J2SDK, trebuie să accesaţi pagina de Internet `http://java.sun.com/j2se`. De acolo, puteţi descărca produsul compatibil cu sistemul de operare pe care îl utilizaţi. Tot de acolo veţi putea descărca şi documentaţia J2SDK.

Digitized by Google

Procedura de instalare a platformei J2SDK crează o ierarhie de directoare care va fi detaliată în cele ce urmează. Pentru a putea verifica autenticitatea informaţiilor prezentate, este necesar să instalaţi J2SDK pe calculatorul dumneavoastră.

În urma instalării, distribuţia Java 2 SDK conţine:

- aplicaţii utilitare pentru dezvoltarea de programe Java (în subdirectorul `bin`). Aceste unelte ajută programatorul să creeze, să execute, să verifice şi să documenteze programe scrise în limbajul Java;
- mediul de execuţie (în subdirectorul `jre`), format din maşina virtuală Java, biblioteca de clase şi alte fişiere necesare execuţiei de programe scrise în limbajul Java. Acest mediu reprezintă o implementare internă a mediului de execuţie Java şi nu trebuie confundat cu *Java 2 Runtime Environment* (*J2RE* - paragraful *1.5.4*). Instalarea J2SDK are ca rezultat şi instalarea J2RE (pentru a verifica acest lucru în cazul sistemului de operare Windows, accesaţi directorul `JavaSoft`, aflat în directorul `Program Files`, în cazul unei instalări standard);
- alte biblioteci (în subdirectorul `lib`), necesare uneltelor de dezvoltare pentru a funcţiona corect;
- aplicaţii demonstrative (în subdirectorul `demo`), ca exemple de programare în limbajul Java;
- fişiere *header C* (în subdirectorul `include`), pentru uz intern;
- codul sursă (în fişierul `src.jar`), pentru o parte a claselor care formează API-ul Java. Acest cod sursă este oferit de Sun doar în scopuri informative, pentru a ajuta programatorii să înveţe limbajul de programare Java. Codul dependent de platformă nu este introdus în această arhivă de fişiere sursă. Pentru a vizualiza fişierele, trebuie să dezarhivaţi fişierul, folosind programul `jar` aflat în distribuţia J2SDK. Comanda trebuie executată în linie de comandă, în directorul în care se află fişierul `src.jar` şi arată astfel:
  `jar xvf src.jar.`
  În urma executării acestei comenzi se va crea un director `src`, care va conţine fişierele sursă.

J2SDK reprezintă primul instrument de dezvoltare de aplicaţii care suportă noile versiuni ale limbajului Java, adesea cu şase luni înaintea apariţiei altui software de dezvoltare Java (de exemplu mediile de programare Symantec Visual Cafe, Borland JBuilder, Metrowerks CodeWarrior, etc).

Digitized by Google

1.5.4 J2RE (Java 2 Runtime Environment, Standard Edition)

Java 2 Runtime Environment este produsul prin intermediul căruia se pot executa aplicaţii scrise în limbajul Java. El este destinat celor care doresc să poată rula aplicaţii Java, fără a avea nevoie de întreg mediul de dezvoltare. Ca şi J2SDK, J2RE conţine maşina virtuală Java (JVM) şi biblioteca de clase care formează Java 2 API. Spre deosebire de J2SDK, J2RE nu conţine unelte de dezvoltare, cum ar fi compilatorul.

J2RE este un produs care poate fi instalat separat, dar şi ca o componentă a J2SDK. J2RE poate fi obţinut la adresa:

`http://java.sun.com/j2se/1.4/jre/index.html`

Produsul poate fi redistribuit gratuit împreună cu o aplicaţie dezvoltată cu J2SDK, astfel încât utilizatorii respectivei aplicaţii să aibă o platformă Java pe care să o poată executa.

Sun oferă produsul J2RE pentru următoarele platforme: Windows, Linux şi Solaris. Există o excepţie în cazul Windows: J2RE nu funcţionează pe platforma Windows NT 3.51.

1.5.5 Documentaţia J2SE (Java 2 Standard Edition Documentation)

Documentaţia J2SE este organizată pe mai multe categorii:

- tutorialul Java ("The Java Tutorial"), ghidul programatorilor Java, cuprinzând sute de exemple funcţionale;
- documentaţia J2SDK;
- specificaţii ale limbajului Java şi ale maşinii virtuale JVM.

Toate aceste documentaţii pot fi obţinute de la adresa:

`http://java.sun.com/docs/index.html`

1.5.6 Licenţa de utilizare a platformei J2SE

Marea majoritate a produselor Sun pot fi obţinute şi folosite gratuit, chiar şi pentru uz comercial. J2SDK şi J2RE intră şi ele în această categorie. Mai mult decât atât, ele pot fi redistribuite gratuit, cu condiţia de a fi însoţite de o aplicaţie. Fiecare produs are propria lui licenţă de utilizare, deci, dacă doriţi amănunte despre acest lucru, puteţi verifica separat licenţa pentru fiecare produs.

Digitized by Google

Rezumat

Deşi aţi parcurs doar primul capitol al cărţii, aţi acumulat totuşi o serie de informaţii despre istoria limbajului Java, principalele versiuni ale limbajului Java existente pe piaţă la ora actuală, şi, nu în ultimul rând, v-aţi familiarizat cu terminologia specifică. Din moment ce noţiunile de `JDK`, `J2SE`, `javac`, `java` vă sunt clare, putem trece la următoarea etapă a iniţierii în tainele acestui limbaj. Pentru aceasta trebuie să vă instalaţi mediul de programare Java şi pe calculatorul dumneavoastră. În momentul în care toate uneltele de dezvoltare a aplicaţiilor Java (editoare, compilatoare, interpretoare) sunt la îndemâna dumneavoastră, vom putea realiza primele programe Java.

Noţiuni fundamentale

API: set de clase utile oferite împreună cu limbajul Java, în vederea reutilizării lor de către programatori;

J2RE: produs ce cuprinde toate aplicaţiile ce permit rularea programelor Java. Reprezintă o componentă a `J2SDK;`

J2SDK: implementare a platformei abstracte `J2SE`, cu ajutorul căreia se pot crea aplicaţii Java;

J2SE: platformă abstractă pentru crearea de aplicaţii Java de tipul *client-side;*

`java`: interpretorul Java ce rulează aplicaţiile Java compilate cu `javac` sau `jikes;`

`javac`: compilator Java realizat de Sun Microsystems;

JDK: vechea denumire a `J2SDK`, utilizată până în decembrie 1998;

`jikes`: compilator Java realizat de IBM;

JVM: maşina virtuală Java, ce interpretează fişierele de tip `.class.`

Erori frecvente

1. `J2SDK` este confundat deseori cu `J2SE`. Deşi diferenţa dintre ele nu este foarte uşor de sesizat, totuşi ea există. `J2SE` reprezintă o platformă abstractă (adică o specificaţie, un standard) în timp ce `J2SDK` este o implementare a platformei abstracte, este unealta cu ajutorul căreia se crează programele Java obişnuite. Până în decembrie 1998, `J2SDK` a fost cunoscut sub denumirea `JDK`.

Digitized by Google

2. Noţiuni fundamentale de programare în Java

Nu sunt nici deosebit de inteligent
şi nici nu am vreun talent
remarcabil.
Sunt doar foarte, foarte curios.

Albert Einstein

Prima parte a cărţii va prezenta, pe parcursul a opt capitole, caracteristicile esenţiale ale limbajului Java. Nu vom include aici descrierea bibliotecilor suplimentare care nu sunt esenţiale pentru înţelegerea limbajului, cum ar fi clasele grafice (AWT, Swing) sau *applet*-urile. De asemenea, tehnologii J2EE avansate, precum *JNI*, *RMI*, *JDBC*, nu sunt prezente nici ele, deoarece descrierea fiecăreia ar necesita cel puţin un volum separat.

După ce în primul capitol aţi făcut cunoştinţă cu universul Java şi v-aţi familiarizat cu terminologia specifică, capitolul de faţă va începe prezentarea limbajului Java prin discutarea tipurilor primitive, a operaţiilor elementare, a instrucţiunilor de decizie şi iterative, a metodelor etc.

În acest capitol vom afla despre:

- Tipurile primitive ale limbajului Java, inclusiv operaţiile care se pot realiza cu variabile de tipuri primitive;

- Cum sunt implementate instrucţiunile de decizie (condiţionale) şi cele iterative în Java;

- Noţiuni elementare despre metode şi supraîncărcarea metodelor.

Digitized by Google

2.1 Mediul de lucru Java

Cum sunt introduse, compilate şi rulate programele Java? Răspunsul depinde, evident, de platforma concretă pe care lucraţi. Anexa A vă pune la dispoziţie detalii referitoare la modul de instalare a mediului de lucru Java pentru diverse sisteme de operare.

Aşa cum am arătat deja în primul capitol, codul sursă Java este conţinut în fişiere text care au extensia `.java`. Compilatorul, care este de obicei `javac` sau `jikes`[1], compilează programul şi generează fişiere `.class` care conţin *bytecode*. *Bytecode* este un limbaj intermediar *portabil* care este interpretat de către maşina virtuală Java, lansată prin comanda `java`. Detalii despre compilatoarele şi interpretorul (maşina virtuală) Java, inclusiv modul de utilizare, se află tot în anexa A. Pentru programele Java, datele de intrare pot să provină din diverse surse:

- de la terminal, numit aici *standard input*;
- parametri suplimentari precizaţi la invocarea programului - parametri în linia de comandă;
- o componentă GUI (*Graphical User Interface*);
- un fişier.

2.2 Primul program Java

În această secţiune vom vedea Java în acţiune, prin crearea şi rularea primei aplicaţii scrisă în acest limbaj de programare. Să începem prin a examina programul simplu din **Listing 2.1**. Acest program tipăreşte un scurt mesaj pe ecran. Numerele din stânga fiecărei linii nu fac parte din program. Ele sunt furnizate doar pentru o mai uşoară referire a secvenţelor de cod.

Listing 2.1: Un prim program simplu

```
//Primul program
public class FirstProgram
{
    public static void main(String[] args)
    {
        System.out.println("Primul meu program Java");
```

[1] `javac` este compilatorul realizat de Sun, `jikes` este compilatorul realizat de IBM şi este preferat de mulţi programatori deoarece este mult mai rapid.

Digitized by Google

```
7       }
8 }
```

Transpuneţi programul într-un fişier cu numele `FirstProgram.java`[2] după care compilaţi-l şi rulaţi-l. Acest lucru se realizează astfel:

- executaţi comanda următoare în directorul în care este salvat fişierul sursă `FirstProgram.java` (folosiţi un *command prompt* de tipul Windows MS-DOS Prompt):

  ```
  javac FirstProgram.java
  ```

- dacă execuţia comezii anterioare s-a realizat fără erori, executaţi comanda următoare în acelaşi director:

  ```
  java FirstProgram
  ```

Rezultatul execuţiei programului este afişarea mesajului `"Primul meu program Java"`. Este important de reţinut că Java este *case-sensitive*, ceea ce înseamnă că face deosebirile între literele mari şi mici.

2.2.1 Comentarii

În Java există trei tipuri de comentarii. Prima formă, care este moştenită de la limbajul C începe cu `/*` şi se termină cu `*/`. Iată un exemplu:

```
/* Acesta este un comentariu
pe doua linii */
```

Comentariile nu pot fi imbricate, deci nu putem avea un comentariu în interiorul altui comentariu.

Cea de-a doua formă de comentarii este moştenită de la limbajul C++ şi începe cu `//`. Nu există simbol pentru încheiere, deoarece un astfel de comentariu se extinde automat până la sfârşitul liniei curente. Acest comentariu este folosit în linia 1 din **Listing 2.1**.

Cea de-a treia formă este asemănătoare cu prima doar că începe cu `/**` în loc de `/*`. Acestă formă de comentariu este utilizată pentru a furniza informaţii utilitarului `javadoc`, prezentat pe larg în anexa B.

Comentariile au fost introduse pentru a face codul mai lizibil pentru programatori. Un program bine comentat reprezintă un semn al unui bun programator.

[2] Atenţie la literele mari şi mici, deoarece compilatorul Java face distincţie între majuscule şi minuscule!

Digitized by Google

2.2.2 Funcţia `main`

Un program Java constă dintr-o colecţie de clase care interacţionează între ele prin intermediul metodelor. Echivalentul Java al unei proceduri sau funcţii din Pascal sau C este *metoda statică*, pe care o vom descrie puţin mai târziu în acest capitol. Atunci când se execută un program Java, maşina virtuală va căuta şi invoca automat metoda statică având numele `main`. Linia 4 din **Listing 2.1** arată că metoda `main` poate fi opţional invocată cu anumiţi parametri în linia de comandă. Tipul parametrilor funcţiei `main` cât şi tipul funcţiei, `void`, sunt obligatorii.

2.2.3 Afişarea pe ecran

Programul din **Listing 2.1** constă dintr-o singură instrucţiune, aflată la linia 6. Funcţia `println` reprezintă principalul mecanism de scriere în Java, fiind echivalent într-o anumită măsură cu funcţia `writeln` din Pascal sau `printf` din C. În această situaţie se scrie un şir de caractere la fluxul de ieşire standard `System.out`. Vom discuta despre citire/scriere mai pe larg în cadrul capitolului 7. Deocamdată ne mulţumim doar să amintim că aceeaşi sintaxă este folosită pentru a scrie orice fel de entitate, fie că este vorba despre un întreg, real, şir de caractere sau alt tip.

2.3 Tipuri de date primitive

Java defineşte opt tipuri primitive de date, oferind în acelaşi timp o foarte mare flexibilitate în a defini noi tipuri de date, numite *clase*. Totuşi, în Java există câteva diferenţe esenţiale între tipurile de date primitive şi cele definite de utilizator. În această secţiune vom examina tipurile primitive şi operaţiile fundamentale care pot fi realizate asupra lor.

2.3.1 Tipurile primitive

Java are opt tipuri de date primitive prezentate în **Tabela 2.1**.

Cel mai des utilizat este tipul întreg specificat prin cuvântul cheie `int`. Spre deosebire de majoritatea altor limbaje de programare, marja de valori a tipurilor întregi nu este dependentă de maşină[3]. Java acceptă şi tipurile întregi `byte`,

[3] Aceasta înseamnă că indiferent pe ce maşină rulăm programul nostru (fie că este un super-computer dotat cu 64 de procesoare, sau un simplu telefon mobil), variabilele de tip `int` vor fi reprezentate pe 32 de biţi, lucru care nu este valabil pentru alte limbaje de programare cum ar fi C sau C++.

Digitized by Google

`short` şi `long`, fiecare fiind adecvat pentru stocare de date întregi având anumite limite. Numerele reale (virgulă mobilă) sunt reprezentate în Java prin tipurile `float` şi `double`. Tipul `double` are mai multe cifre semnficative, de aceea utilizarea lui este recomandată în locul tipului `float`. Tipul `char` este folosit pentru a reprezenta caractere. Un `char` ocupă în Java 16 biţi (şi nu doar 8, cum s-ar aştepta programatorii C sau Pascal, obişnuiţi cu caractere ASCII) pentru a putea reprezenta toate caracterele din standardul *Unicode*. Acest standard conţine peste 30.000 de caractere distincte care acoperă principalele limbi scrise (inclusiv limbile japoneză, chineză etc.). Prima parte a tabelei *Unicode* este identică cu tabela *ASCII*, deci setul de caractere Unicode este o extensie a mai vechiului set de caractere ASCII. Ultimul tip primitiv al limbajului Java este `boolean`; o variabilă de tip `boolean` poate lua una din valorile de adevăr `true` sau `false`.

Tabela 2.1: Cele 8 tipuri primitive de date în Java

Tip dată	Ce reţine	Valori
byte	întreg pe 8 biţi	-128 la 127
short	întreg pe 16 biţi	-32768 la 32767
int	întreg pe 32 biţi	-2.147.483.648 la 2.147.483.647
long	întreg pe 64 biţi	-2^{63} la $2^{63}-1$
float	virgulă mobilă pe 32 biţi	6 cifre semnificative (10^{-46} la 10^{38})
double	virgulă mobilă pe 64 biţi	15 cifre semnificative (10^{-324} la 10^{308})
char	caracter unicode	
boolean	variabilă booleană	false şi true

2.3.2 Constante

Constantele întregi pot fi reprezentate în bazele 10, 8 sau 16. Notaţia *octală* este indicată printr-un 0 nesemnificativ la început, iar notaţia *hexa* este indicată printr-un 0x sau 0X la început. Iată reprezentarea întregului 37 în câteva moduri echivalente: 37, 045, 0x25. Notaţiile octale şi hexazecimale nu vor fi utilizate în această carte. Totuşi trebuie să fim conştienţi de ele pentru a folosi 0-uri la început doar acolo unde chiar vrem aceasta.

O *constantă caracter* este cuprinsă între apostrofuri, cum ar fi `'a'`. Intern, Java interpretează această constantă ca pe un număr (codul *Unicode* al caracterului respectiv). Ulterior, funcţiile de scriere vor transforma acest număr în caracterul corespunzător. Constantele caracter mai pot fi reprezentate şi în

Digitized by Google

forma:

```
'\uxxxx'.
```

unde `xxxx` este un număr în baza 16 reprezentând codul *Unicode* al caracterului.

Constantele de tip şir de caractere sunt cuprinse între ghilimele, ca în `"Primul meu program Java"`. Există anumite secvenţe speciale, numite *secvenţe escape*, care sunt folosite pentru anumite caractere speciale. Noi vom folosi mai ales

```
'\n', '\\', '\'', '\"',
```

care înseamnă, respectiv, *linie nouă*, *backslash*, *apostrof* şi *ghilimele*.

2.3.3 Declararea şi iniţializarea tipurilor primitive în Java

Orice variabilă Java, inclusiv cele primitive, este declarată prin descrierea numelui, a tipului şi, opţional, a valorii iniţiale. Numele variabilei trebuie să fie un *identificator*. Un identificator poate să conţină orice combinaţie de litere, cifre şi caracterul *underscore* (liniuţa de subliniere "_"). Identificatorii nu pot începe cu o cifră. Cuvintele rezervate, cum ar fi `int` nu pot fi identificatori. Nu pot fi utilizaţi nici identificatorii care deja sunt declaraţi şi sunt vizibili.

Java este *case-sensitive*, ceea ce înseamnă că `sum` şi `Sum` reprezintă identificatori diferiţi. Pe parcursul acestei cărţi vom folosi următoarea convenţie pentru numele variabilelor:

- toate numele de variabilă încep cu literă mică, iar cuvintele noi din cadrul numelui încep cu literă mare. De exemplu: `sumaMaxima`, `nodVizitat`, `numberOfStudents` etc.;
- numele claselor începe cu literă mare. De exemplu: `ArithmeticException`, `FirstProgram`, `BinaryTree` etc.

Pentru detalii complete asupra convenţiilor vă recomandăm anexa B, unde veţi găsi informaţii despre convenţiile de scriere a programelor Java, precum şi despre o aplicaţie foarte utilă oferită de Sun, şi anume `javadoc`, care permite foarte uşor crearea unor documentaţii ale programelor dumneavoastră.

Iată câteva exemple de declaraţii de variabile:

```
int numarElemente;
double mediaGenerala;
int produs = 1, suma = 0;
int produs1 = produs;
```


Digitized by Google

O variabilă este bine să fie declarată imediat înainte de a fi folosită. Asa cum vom vedea mai târziu, locul unde este declarată determină domeniul de vizibilitate şi semnificaţia ei.

Limbajul Java permite şi definirea de constante. Deoarece acestea necesită cunoştinţe mai aprofundate, ne rezumăm la a spune că ele vor fi prezentate în cadrul subcapitolului destinat atributelor statice.

2.3.4 Citire/scriere la terminal

Scrierea la terminal în Java se realizează cu funcţia `println` şi nu pune probleme majore. Lucrurile nu stau deloc la fel cu citirea de la tastatură, care se realizează mult mai anevoios. Acest lucru se datorează în primul rând faptului că programele Java nu sunt concepute pentru a citi de la tastatură. În imensa majoritate a cazurilor, programele Java îşi preiau datele dintr-o *interfaţă grafică* (*Applet-urile*), din forme *HTML (Java Servlets, Java Server Pages)* sau din *fişiere*, fie că sunt fişiere standard (fişiere text, binare etc.) sau colecţii de resurse (*resource bundles*).

Citirea şi scrierea de la consolă sunt realizate prin metodele `readLine`, respectiv `println`. *Fluxul de intrare standard* este `System.in`, iar *fluxul de ieşire standard* este `System.out`.

Mecanismul de bază pentru citirea/scrierea formatată foloseşte tipul `String`, care va fi descris în capitolul următor. La afişare, operatorul + concatenează două `String`-uri. Pentru tipurile primitive, dacă parametrul scris nu este de tip `String` se face o conversie temporară la `String`. Aceste conversii pot fi definite şi pentru obiecte, aşa cum vom arăta mai târziu. Pentru citire se asociază un obiect de tipul `BufferedReader` cu `System.in`. Apoi se citeşte un `String` care va fi ulterior prelucrat. Capitolul 7 vă va edifica mai mult asupra acestor operaţii.

2.4 Operatori de bază

Această secţiune descrie operatorii de bază care sunt disponibili în Java. Aceşti operatori sunt utilizaţi pentru a crea expresii. O constantă sau o variabilă reprezintă o *expresie*, la fel ca şi combinaţiile de constante şi variabile cu operatori. O expresie urmată de simbolul `;` reprezintă o *instrucţiune simplă*.

Digitized by Google

2.4.1 Operatori de atribuire

Programul simplu din **Listing 2.2** ilustrează câţiva operatori Java. *Operatorul de atribuire* este *semnul egal* (=). De exemplu, în linia 12, variabilei a i se atribuie valoarea variabilei b (care în acel moment are valoarea 7). Modificările ulterioare ale variabilei b nu vor afecta variabila a. Operatorii de atribuire pot fi înlănţuiţi ca în următorul exemplu, atribuirile făcându-se (asociindu-se) de la dreapta la stânga:

```
z = y = x = 0;
```

Un alt operator de atribuire este += al cărui mod de utilizare este ilustrat în linia 18. Operatorul += adaugă valoarea aflată la dreapta (operatorului) la variabila din stânga. Astfel, valoarea lui b este incrementată de la 7 la 11. Java oferă şi alţi operatori de atribuire cum ar fi −=, *= şi /= care modifică variabila aflată în partea stângă prin scădere, înmulţire şi respectiv împărţire.

Listing 2.2: Program care ilustrează anumiţi operatori simpli

```
1 /** Utilizarea operatorilor. */
2 public class OperatorTest
3 {
4     public static void main(String[] args)
5     {
6         int a = 5, b = 7, c = 4;
7
8         System.out.println("a=" + a); //a=5
9         System.out.println("b=" + b); //b=7
10        System.out.println("c=" + c); //c=4
11
12        a = b;
13
14        System.out.println("a=" + a); //a=7
15        System.out.println("b=" + b); //b=7
16        System.out.println("c=" + c); //c=4
17
18        b += c;
19
20        System.out.println("a=" + a); //a=7
21        System.out.println("b=" + b); //b=11
22        System.out.println("c=" + c); //c=4
23
24        c = a + b;
25
26        System.out.println("a=" + a); //a=7
27        System.out.println("b=" + b); //b=11
28        System.out.println("c=" + c); //c=18
29
30        c++;
```


Digitized by Google

```

        System.out.println("a=" + a); //a=7
        System.out.println("b=" + b); //b=11
        System.out.println("c=" + c); //c=19

        ++a;

        System.out.println("a=" + a); //a=8
        System.out.println("b=" + b); //b=11
        System.out.println("c=" + c); //c=19

        b = ++a + c++;

        System.out.println("a=" + a); //a=9
        System.out.println("b=" + b); //b=28
        System.out.println("c=" + c); //c=20
    }
}
```

2.4.2 Operatori aritmetici binari

Linia 24 din **Listing 2.2** ilustrează unul dintre operatorii binari tipici ai limbajelor de programare: *operatorul de adunare* (+). Operatorul + are ca efect adunarea conţinutului variabilelor `a` şi `b`; valorile lui `a` şi `b` rămân neschimbate. Valoarea rezultată este atribuită lui `c`. Alţi operatori aritmetici folosiţi în Java sunt: `-`, `*`, `/` şi `%` utilizaţi respectiv pentru *scădere, înmulţire, împărţire* şi *rest*.

Împărţirea a două valori întregi are ca valoare doar partea întreagă a rezultatului. De exemplu `3/2` are valoarea `1`, dar `3.0/2` are valoarea `1.5`, deoarece unul dintre operanzi (3.0) a fost o valoare în virgulă mobilă, şi astfel rezultatul întregii expresii este convertit la virgulă mobilă.

Aşa cum este şi normal, adunarea şi scăderea au aceeaşi prioritate. Această prioritate este mai mică decât cea a grupului format din înmulţire, împărţire şi rest; astfel `1 + 2 * 3` are valoarea `7`. Toţi aceşti operatori sunt evaluaţi de la stânga la dreapta (astfel `3 - 2 - 2` are valoarea `-1`). Toţi operatorii aritmetici au o anumită prioritate şi o anumită asociere (fie de la stânga la dreapta, fie de la dreapta la stânga, descrisă în **Tabela** 2.2[4]).

2.4.3 Operatori aritmetici unari

În plus faţă de operatorii aritmetici binari care necesită doi operanzi, Java dispune şi de *operatori unari* care necesită doar un singur operand. Cel mai

[4]Prin asociere înţelegem ordinea de evaluare într-o expresie care conţine operatori de acelaşi tip şi nu are paranteze (de exemplu, $a - b - c$).

Digitized by Google

cunoscut operator unar este *operatorul minus* (−) care returnează operandul cu semn opus. Astfel, `-x` este opusul lui `x`.

Java oferă de asemenea *operatorul de autoincrementare* care adaugă 1 la valoarea unei variabile, notat prin ++, şi *operatorul de autodecrementare* care scade 1 din valoarea variabilei, notat cu −−. Un caz banal de utilizare a acestor operatori este exemplificat în liniile 30 şi 36 din **Listing 2.2**. În ambele cazuri operatorul ++ adaugă 1 la valoarea variabilei. În Java, ca şi în C, orice expresie are o valoare. Astfel, un operator aplicat unei variabile generează o expresie cu o anumită valoare. Deşi faptul că variabila este incrementată înainte de execuţia următoarei instrucţiuni este garantat, se pune întrebarea: "Care este valoarea expresiei de autoincrementare dacă ea este utilizată în cadrul unei alte expresii?"

În acest caz, locul unde se plasează operatorul ++ este esenţial. Semnficaţia lui `++x` este că valoarea expresiei este egală cu noua valoare a lui `x`. Acest operator este numit *incrementare prefixată*. În mod analog, `x++` înseamnă că valoarea expresiei este egală cu valoarea originală a lui `x`. Acesta este numit *incrementare postfixată*. Aceste trăsături sunt exemplificate în linia 42 din **Listing 2.2**. Atât `a`, cât şi `c` sunt incrementate cu 1, iar `b` este obţinut prin adunarea valorii incrementate a lui `a` (care este 9) cu valoarea iniţială a lui `c` (care este 19).

2.4.4 Conversii de tip

Operatorul conversie de tip, numit în jargonul programatorilor şi *operatorul de cast*, este utilizat pentru a genera o variabilă temporară de un nou tip. Să considerăm, de exemplu, secvenţa de cod:

```
double rest;
int x = 2;
int y = 5;
rest = x / y; //probabil incorect!
```

La efectuarea operaţiei de împărţire, atât `x` cât şi `y` fiind numere întregi, se va realiza o împărţire întreagă şi se obţine 0. Întregul 0 este apoi convertit implicit la `double` astfel încât să poată fi atribuit lui `rest`. Probabil că intenţia noastră era aceea de a atribui lui `rest` valoarea 0.4. Soluţia este de a converti temporar pe `x` sau pe `y` la `double`, pentru ca împărţirea să se realizeze în virgulă mobilă. Acest lucru se poate obţine astfel:

```
rest = (double) x / y;
```

De remarcat că nici `x` şi nici `y` nu se schimbă. Se crează o variabilă temporară fără nume, având valoarea 2.0, iar valoarea ei este utilizată pentru a efectua

Digitized by Google

împărţirea. Operatorul de conversie de tip are o prioritate mai mare decât operatorul de împărţire, de aceea conversia de tip se efectuează înainte de a se efectua împărţirea.

2.5 Instrucţiuni condiţionale

Această secţiune este dedicată instrucţiunilor care controlează fluxul de execuţie al programului: instrucţiunile condiţionale şi iteraţia.

2.5.1 Operatori relaţionali

Testul fundamental care poate fi realizat asupra tipurilor primitive este *comparaţia*. Comparaţia se realizează utilizând *operatorii de egalitate/inegalitate* şi *operatorii de comparaţie* (<, > etc.). În Java, operatorii de egalitate/inegalitate sunt == respectiv `!=`. De exemplu,

```
exprStanga == exprDreapta
```

are valoarea `true` dacă `exprStanga` şi `exprDreapta` sunt egale, altfel are valoarea `false`. Analog, expresia:

```
exprStanga != exprDreapta
```

are valoarea `true` dacă `exprStanga` şi `exprDreapta` sunt diferite; altfel are valoarea `false`.

Operatorii de comparaţie sunt <, <=, >, >= iar semnificaţia lor este cea naturală pentru tipurile fundamentale. Operatorii de comparaţie au prioritate mai mare decât operatorii de egalitate. Totuşi, ambele categorii au prioritate mai mică decât operatorii aritmetici, dar mai mare decât operatorii de atribuire. Astfel, veţi constata că în cele mai multe cazuri folosirea parantezelor nu va fi necesară. Toţi aceşti operatori se evaluează de la stânga la dreapta, dar cunoaşterea acestui lucru nu ne foloseşte prea mult. De exemplu, în expresia `a < b < 6`, prima comparaţie generează o valoare booleană, iar a doua expresie este greşită, deoarece operatorul < nu este definit pentru valori booleene. Paragraful următor descrie cum se poate realiza acest test în mod corect.

2.5.2 Operatori logici

Java dispune de *operatori logici* care sunt utilizaţi pentru a simula operatorii *and, or* şi *not* din algebra *Booleană*. Aceşti operatori sunt referiţi uneori şi sub numele de *conjuncţie, disjuncţie* şi, respectiv, *negare*, simbolurile corespunzătoare fiind `&&`, `||` şi `!`. Implementarea corectă a testului din paragraful anterior este:

Digitized by Google

```
(a < b) && (b < 6)
```

Prioritatea conjuncţiei şi a disjuncţiei este suficient de mică faţă de prioritatea celorlalţi operatori din expresie pentru ca parantezele să nu fie necesare. `&&` are prioritate mai mare decât `||`, iar `!` are aceeaşi prioritate cu alţi operatori unari (`++`, `--`, vezi **Tabela 2.2**).

Tabela 2.2: Operatori Java listaţi în ordinea priorităţii

Categorie	Exemple	Asociere
Operatori pe referinţe	. [ ]	Stânga la dreapta
Unari	++ – ! - (tip)	Dreapta la stânga
Multiplicativi	* / %	Stânga la dreapta
Aditivi	+ -	Stânga la dreapta
Shiftare (pe biţi)	<< >> >>>	Stânga la dreapta
Relaţionali	< <= > >= instanceof	Stânga la dreapta
Egalitate	== !=	Stânga la dreapta
AND pe biţi	&	Stânga la dreapta
XOR pe biţi	^	Stânga la dreapta
OR pe biţi	|	Stânga la dreapta
AND logic	&&	Stânga la dreapta
OR logic	||	Stânga la dreapta
Condiţional	?:	Dreapta la stânga
Atribuire	= *= /= %= += -=	Dreapta la stânga

O regulă importantă este că operatorii `&&` şi `||` folosesc evaluarea booleană scurtcircuitată[5]. Aceasta înseamnă că dacă rezultatul poate fi determinat evaluând prima expresie, a doua expresie nu mai este evaluată. De exemplu, în expresia:

```
x != 0 && 1/x != 3
```

dacă `x` este 0, atunci prima jumătate este `false`. Aceasta înseamnă că rezultatul conjuncţiei va fi fals, deci a doua expresie nu mai este evaluată. Acesta este un detaliu important, deoarece împărţirea la 0 ar fi generat un comportament eronat.

[5]Numită uneori şi evaluare booleană parţială.

Digitized by Google

2.5.3 Operatori la nivel de bit

Operatorii la nivel de bit pe care limbajul Java îi pune la dispoziţia programatorului se împart în două grupe:

- operatori de deplasare (*shiftare*) : >> << >>>
- operatori logici : & | ^ ~

Fie că fac parte din prima sau a doua categorie, operatorii la nivel de bit nu pot fi utilizaţi decât în cazul numerelor întregi (variabile de tipul `byte`, `short`, `int` şi `long`). De fapt, operatorii se aplică reprezentării binare a numerelor implicate. Cu alte cuvinte, dacă avem operaţia `5 & 3`, aceasta înseamnă că de fapt operatorul `&` se aplică reprezentării în baza 2 a numerelor 5 şi 3, adică `101` şi `11`.

Prima grupă de operatori la nivel de bit, cea a operatorilor de deplasare, execută manipularea biţilor primului operand, deplasându-i la stânga sau la dreapta, în funcţie de tipul operatorului (vezi **Tabela 2.3**).

Tabela 2.3: Operatori la nivel de bit pentru deplasare

Operator	Utilizare	Descriere
>>	op1 >> op2	deplasarea biţilor lui op1 la dreapta cu op2 poziţii
<<	op1 << op2	deplasarea biţilor lui op1 la stânga cu op2 poziţii
>>>	op1 >>> op2	deplasarea biţilor lui op1 la dreapta cu op2 poziţii, op1 fiind considerat `unsigned`

Fiecare operator deplasează biţii operandului din stânga cu un număr de poziţii indicat de operandul din dreapta. Deplasarea se face în sensul indicat de operator (>> la dreapta, iar << la stânga). De exemplu, următoarea instrucţiune deplasează biţii numărului 5 la dreapta cu o poziţie:

```
5 >> 1;
```

Reprezentarea binară a numărului 5 este 101. Prin deplasarea biţilor la dreapta cu o poziţie se obţine numărul care are reprezentarea binară 10, deoarece primul 1 (cel din stânga) trece cu o poziţie mai la dreapta, deci în locul lui 0, care trece cu o poziţie mai la dreapta, deci în locul celei de-a doua cifre 1,

Digitized by Google

care se pierde. Biţii din stânga care au rămas descompletaţi se completează cu 0, ceea ce conduce la rezultatul final 010. Astfel, am obţinut numărul 2 în reprezentarea zecimală.

În concluzie, `5 >> 1 = 2`. După cum se poate observa, pentru numerele pozitive, deplasarea la dreapta cu `op2` biţi este echivalentă cu împărţirea lui `op1` la $2^{\texttt{op2}}$. Pentru exemplul nostru, $5/2^1 = 2$.

Analog se întâmplă şi în cazul deplasării la stânga. Instrucţiunea următoare deplasează biţii numărului 5, la stânga, cu o poziţie:

```
5 << 1;
```

Rezultatul deplasării biţilor 101 la stânga cu o poziţie este 1010, adică numărul 10 în reprezentare zecimală. Acest lucru este echivalent cu înmulţirea cu 2^{op2}. În cazul nostru, $5 * 2^1 = 10$.

Cel de-al treilea tip de deplasare a biţilor, >>>, este asemănător cu >>, cu specificarea că numărul `op1` este considerat fără semn (unsigned).

Cea de-a doua categorie de operatori la nivel de bit realizează operaţii logice ŞI, SAU, SAU EXCLUSIV (XOR) şi NEGARE:

Tabela 2.4: Operatori pentru operaţii logice la nivel de bit

<table>
<tr><th>Operator</th><th>Utilizare</th><th>Descriere</th></tr>
<tr><td>&</td><td>op1 & op2</td><td>ŞI la nivel de biţi</td></tr>
<tr><td>|</td><td>op1 | op2</td><td>SAU la nivel de biţi</td></tr>
<tr><td>^</td><td>op1 ^ op2</td><td>SAU EXCLUSIV la nivel de biţi</td></tr>
<tr><td>~</td><td>~op1</td><td>NEGARE la nivel de biţi</td></tr>
</table>

Să presupunem că avem două numere, 5 şi 3, cărora dorim să le aplicăm operaţii logice la nivel de biţi. Operaţiile logice la nivel de biţi constau în aplicarea operaţiei respective perechilor de biţi de pe poziţii egale în cele două numere (cu excepţia operaţiei de negare care este unară). În situaţia în care numerele nu au reprezentarea binară de aceeaşi lungime, reprezentarea mai scurtă este completată cu zerouri nesemnificative (inserate în faţa reprezentării), până se obţin dimensiuni egale.

Prezentate pe scurt, operaţiile logice pe perechi de biţi se realizează pe baza următorului alogritm:

Digitized by Google

- expresia `b1 & b2` are valoarea 1, dacă şi numai dacă `b1` şi `b2` au valoarea 1 (`b1` şi `b2` sunt 2 biţi, care pot lua, evident, doar valorile 0 şi 1). Altfel, expresia are valoarea 0;

- expresia `b1 | b2` are valoarea 0, dacă şi numai dacă `b1` şi `b2` au valoarea 0. Altfel, expresia are valoarea 1;

- expresia `b1 ^ b2` are valoarea 1, dacă şi numai dacă unul dintre operanzi este 0, iar celălalt 1. Altfel, expresia are valoarea 0;

- expresia `~b1` reprezintă negarea lui `b1`, deci are valoarea 1, dacă `b1` este 0, sau are valoarea 0, dacă `b1` este 1.

Iată câteva exemple de utilizare:

- `5 & 3 = 1`

```
   101   // = 5
&  011   // = 3
------
   001   // = 1
```

- `5 | 3 = 7`

```
   101   // = 5
|  011   // = 3
------
   111   // = 7
```

- `5 ^ 3 = 6`

```
   101   // = 5
^  011   // = 3
------
   110   // = 6
```

- `~5 = -6`

```
~  0...0101   // = 5
   --------
   1...1010   // = -6
```


Digitized by Google

Această operaţie necesită anumite precizări. După cum am văzut anterior, un număr de tip `int` este reprezentat pe 32 de biţi. Cu alte cuvinte, reprezentarea numărului 5 în baza 2, are de fapt 32 de cifre binare, dintre care 29 sunt zerouri nesemnificative (cele din faţă), pe care le-am omis până acum din motive de spaţiu. La o operaţie de negare, toate zerourile nesemnificative ale reprezentării binare devin 1 şi capătă astfel importanţă (primul bit este bitul de semn, care se modifică şi el). Pentru a înţelege mai bine modul în care se reprezintă numerele întregi în memorie vă recomandăm lucrarea [Boian].

2.5.4 Instrucţiunea `if`

Instrucţiunea `if` este instrucţiunea fundamentală de *decizie*. Forma sa simplă este:

```
if (expresie)
    instructiune

urmatoareaInstructiune
```

Dacă `expresie` are valoarea `true` atunci se execută `instructiune`; în caz contrar, `instructiune` nu se execută. După ce instrucţiunea `if` se încheie fără incidente (vezi cazul excepţiilor netratate, în capitolul 6), controlul este preluat de `urmatoareaInstructiune`.

Opţional, putem folosi instrucţiunea `if-else` după cum urmează:

```
if (expresie)
    instructiune1
else
    instructiune2

urmatoareaInstructiune
```

În acest caz, dacă `expresie` are valoarea `true`, atunci se execută `instructiune1`; altfel se execută `instructiune2`. În ambele cazuri controlul este apoi preluat de `urmatoareaInstructiune`. Iată un exemplu:

```
System.out.println("1/x este: ");
if (x != 0)
    System.out.print( 1/x );
else
    System.out.print("Nedefinit");

System.out.println();
```

De reţinut că doar o *singură* instrucţiune poate exista pe ramura de `if` sau de `else` indiferent de cum indentaţi codul. Iată două erori frecvente pentru

Digitized by Google

începători:

```
if (x == 0) ; //instructiune vida!!!
    System.out.println("x este 0");
else
    System.out.print("x este");
    System.out.println(x); //instructiune in afara clauzei else
```

Prima greşeală constă în a pune `;` după `if`. Simbolul `;` reprezintă în sine *instrucţiunea vidă*; ca o consecinţă, acest fragment de cod nu va fi compilabil (`else` nu va fi asociat cu nici un `if`). După ce am corectat această eroare, rămânem cu o eroare de logică: ultima linie de cod nu face parte din `if`, deşi acest lucru este sugerat de indentare. Pentru a rezolva această problemă vom utiliza un bloc în care grupăm o secvenţă de instrucţiuni printr-o pereche de acolade:

```
if (x == 0)
{
    System.out.println("x este 0");
}
else
{
    System.out.print("x este");
    System.out.println(x);
}
```

Practic, prin cuprinderea mai multor instrucţiuni între acolade, creăm o singură instrucţiune, numită *instrucţiune compusă*. Aşadar, `instructiune1` (ca şi `instructiune2`) poate fi o instrucţiune simplă sau o instrucţiune compusă. Instrucţiunile compuse sunt formate dintr-o succesiune de instrucţiuni (simple sau compuse) cuprinse între acoloade. Această definiţie permite imbricarea instrucţiunilor compuse în cadrul altor instrucţiuni compuse. În exemplul anterior, pe ramura `if` avem o instrucţiune compusă formată doar dintr-o singură instrucţiune simplă:

```
{
    System.out.println("x este 0");
}
```

Am folosit aici o instrucţiune compusă deşi, fiind vorba de o singură instrucţiune simplă, acest lucru nu era absolut necesar. Totuşi, această practică îmbunătăţeşte foarte mult lizibilitatea codului şi vă invităm şi pe dumneavoastră să o adoptaţi.

Instrucţiunea `if` poate să facă parte dintr-o altă instrucţiune `if` sau `else`, la fel ca şi celelalte instrucţiuni de control prezentate în continuare în această secţiune.

Digitized by Google

2.5.5 Instrucţiunea `while`

Java, ca şi Pascal sau C, dispune de trei *instrucţiuni de ciclare* (repetitive): instrucţiunea *while,* instrucţiunea *do* şi instrucţiunea *for*. Sintaxa instrucţiunii `while` este:

```
while (expresie)
    instructiune

urmatoareaInstructiune
```

Observaţi că, la fel ca şi la instrucţiunea `if`, nu există `;` în sintaxă. Dacă apare un `;` după `while`, va fi considerat ca instrucţiune vidă.

Cât timp `expresie` este `true` se execută `instructiune`; apoi `expresie` este evaluată din nou. Dacă `expresie` este `false` de la bun început, atunci `instructiune` nu va fi executată niciodată. În general, `instructiune` face o acţiune care ar putea modifica valoarea lui `expresie`; altfel, ciclarea s-ar putea produce la infinit. Când execuţia instrucţiunii `while` se încheie, controlul este preluat de `urmatoareaInstructiune`.

2.5.6 Instrucţiunea `for`

Instrucţiunea `while` ar fi suficientă pentru a exprima orice fel de ciclare. Totuşi, Java mai oferă încă două forme de a realiza ciclarea: instrucţiunea `for` şi instrucţiunea `do`. Instrucţiunea `for` este utilizată în primul rând pentru a realiza iteraţia. Sintaxa ei este:

```
for (initializare; test; actualizare)
    instructiune

urmatoareaInstructiune
```

În această situaţie, `initializare`, `test` şi `actualizare` sunt toate expresii, şi toate trei sunt opţionale. Dacă `test` lipseşte, valoarea sa implicită este `true`. După paranteza de închidere nu se pune `;`.

Instrucţiunea `for` se execută realizând mai întâi `initializare`. Apoi, cât timp `test` este `true` se execută `instructiune`, iar apoi se execută `actualizare`. Dacă `initializare` şi `actualizare` sunt omise, instrucţiunea `for` se va comporta exact ca şi instrucţiunea `while`.

Avantajul instrucţiunii `for` constă în faptul că se poate vedea clar marja pe care iterează variabilele contor.

Următoarea secvenţă de cod afişează primele 100 numere întregi pozitive:

```
for (int i = 1; i < 100; ++i)
{
```


Digitized by Google

```
    System.out.println(i);
}
```

Acest fragment ilustrează şi practica obişnuită pentru programatorii Java (şi C++) de a declara un contor întreg în secvenţa de iniţializare a ciclului. Durata de viaţă a acestui contor se extinde doar în interiorul ciclului.

Atât `initializare` cât şi `actualizare` pot folosi operatorul *virgulă* pentru a permite *expresii multiple*. Următorul fragment ilustrează această tehnică frecvent folosită:

```
for (i = 0, sum = 0; i <= n; i++, sum += n)
{
    System.out.println(i + "\t" + sum);
}
```

Ciclurile pot fi imbricate la fel ca şi instrucţiunile `if`. De exemplu, putem găsi toate perechile de numere mici a căror sumă este egală cu produsul lor (cum ar fi 2 şi 2, a căror sumă şi produs este 4) folosind secvenţa de cod de mai jos:

```
for (int i = 1; i <= 10; i++)
{
    for (int j = 1; j <= 10; j++)
    {
        if (i + j == i * j)
        {
            System.out.println(i + ", " + j);
        }
    }
}
```

2.5.7 Instrucţiunea `do`

Instrucţiunea `while` realizează un test repetat. Dacă testul este `true` atunci se execută instrucţiunea din cadrul ei. Totuşi, dacă testul iniţial este `false`, instrucţiunea din cadrul ciclului nu este executată niciodată. În anumite situaţii avem nevoie ca instrucţiunile din ciclu să se execute cel puţin o dată. Acest lucru se poate realiza utilizând instrucţiunea `do`. Instrucţiunea `do` este asemănătoare cu instrucţiunea `while`, cu deosebirea că testul este realizat după ce instrucţiunile din corpul ciclului se execută. Sintaxa sa este:

```
do
    instructiune
while (expresie);

urmatoareaInstructiune;
```


Digitized by Google

Remarcaţi faptul că instrucţiunea `do` se termină cu `;`. Un exemplu tipic în care se utilizează instrucţiunea `do` este dat de fragmentul de (pseudo-) cod de mai jos:

```
do
{
    afiseaza mesaj;
    citeste data;
}
while (data nu este corecta);
```

Instrucţiunea `do` este instrucţiunea de ciclare cel mai puţin utilizată. Totuşi, când vrem să executăm cel puţin o dată instrucţiunile din cadrul ciclului, şi `for` este incomod de utilizat, atunci `do` este alegerea potrivită.

2.5.8 Instrucţiunile `break` şi `continue`

Instrucţiunile `for` şi `while` au condiţia de terminare *înaintea* instrucţiunilor care se repetă. Instrucţiunea `do` are condiţia de terminare *după* instrucţiunile care se repetă. Totuşi, în anumite situaţii, s-ar putea să fie nevoie să întrerupem ciclul chiar în mijlocul instrucţiunilor care se repetă. În acest scop, se poate folosi instrucţiunea `break`. De obicei, instrucţiunea `break` apare în cadrul unei instrucţiuni `if`, ca în exemplul de mai jos:

```
while (...)
{
    ...
    if (conditie)
    {
        break;
    }
    ...
}
```

În cazul în care sunt două cicluri imbricate, instrucţiunea `break` părăseşte doar ciclul cel mai din interior. Dacă există mai mult de un ciclu care trebuie terminat, `break` nu va funcţiona corect, şi mai mult ca sigur că aţi proiectat prost algoritmul. Totuşi, Java oferă aşa numitul `break` etichetat. În acest caz, o anumită instrucţiune de ciclare este etichetată şi instrucţiunea `break` poate fi aplicată acelei instrucţiuni de ciclare, indiferent de numărul de cicluri imbricate. Iată un exemplu:

```
eticheta:
    while (...)
    {
        while(...)
        {
```


Digitized by Google

```
            ...
            if (conditie)
            {
                break eticheta;
            }
        }
    }
//controlul programului trece aici dupa executia lui break
```

În anumite situaţii dorim să renunţăm la execuţia iteraţiei curente din ciclu şi să trecem la următoarea iteraţie a ciclului. Acest lucru poate fi realizat cu instrucţiunea `continue`. Ca şi `break`, instrucţiunea `continue` este urmată de `;` şi se aplică doar ciclului cel mai interior în cazul ciclurilor imbricate. Următorul fragment tipăreşte primele 100 de numere întregi, cu excepţia celor divizibile cu 10:

```
for (int i = 1; i <= 100; i++)
{
    if (i % 10 == 0)
    {
        continue;
    }

    System.out.println(i);
}
```

Desigur, că exemplul de mai sus poate fi implementat şi utilizând un `if` simplu. Totuşi instrucţiunea `continue` este adeseori folosită pentru a evita imbricări complicate de tip `if-else` în cadrul ciclurilor.

2.5.9 Instrucţiunea `switch`

Instrucţiunea `switch` este numită uneori şi *instrucţiune de selecţie*; ea are rolul de a selecta dintre mai multe secvenţe de cod, una care va fi executată, funcţie de valoarea unei expresii întregi. Forma sa este:

```
switch (expresie-selectare)
{
    case valoare-intreaga1:
            instructiune;
            break;
    case valoare-intreaga2:
            instructiune;
            break;

    //...

    case valoare-intreagaN:
```


Digitized by Google

```
            instructiune;
            break;

    default:
            instructiune;
}
```

`expresie-selectare` este o expresie care produce o valoare întreagă. Instrucţiunea `switch` compară valoarea expresiei `expresie-selectare` cu fiecare `valoare-intreaga`. Dacă are loc egalitatea, se execută instrucţiunea corespunzătoare (simplă sau compusă). Dacă nu are loc nici o egalitate se execută instrucţiunea din `default` (alternativa `default` este opţională în cadrul `switch`).

Observaţi că în exemplul de mai sus, fiecare `case` se încheie cu un `break` care are ca efect saltul la sfârşitul instrucţiunii `switch`. Acesta este modul obişnuit de a scrie o instrucţiune de tip `switch`, dar prezenţa instrucţiunii `break` nu este obligatorie. Dacă instrucţiunea `break` lipseşte, atunci se va executa şi codul corespunzător instrucţiunilor `case` următoare până când se întâlneşte un `break`. Deşi de obicei nu ne dorim un astfel de comportament, el poate fi uneori util pentru programatorii experimentaţi.

Pentru a exemplifica instrucţiunea `switch`, programul din **Listing 2.3** crează litere aleator şi determină dacă acestea sunt vocale sau consoane (în limba engleză).

Listing 2.3: Program care exemplifică instrucţiunea `switch`

```
//VowelsAndConsonants.java
// Program demonstrativ pentru instructiunea switch
public class VowelsAndConsonants
{
  public static void main(String[] args)
  {
    for (int i = 0; i < 100; i++)
    {
      char c = (char) (Math.random() * 26 + 'a');
      System.out.print(c + ": ");
      switch(c)
      {
        case 'a':
        case 'e':
        case 'i':
        case 'o':
        case 'u':
          System.out.println("vocala");
          break;
        case 'y':
        case 'w':
```


Digitized by Google

```
                System.out.println("Uneori vocale ");
                //doar in limba engleza!
                break;
            default:
                System.out.println("consoana");
        } //switch
      } //for
    } //main
} //class
```

Funcţia `Math.random()` generează o valoare în intervalul [0,1). Prin înmulţirea valorii returnate de această funcţie cu numărul de litere din alfabet (26 litere) se obţine un număr în intervalul [0,26). Adunarea cu prima literă (`'a'`, care are de fapt valoarea 97, codul ASCII al literei 'a') are ca efect transpunerea în intervalul [97,123). În final se foloseşte operatorul de conversie de tip pentru a trunchia numărul la o valoare din mulţimea 97, 98, ..., 122, adică un cod ASCII al unui caracter din alfabetul englez.

2.5.10 Operatorul condiţional

Operatorul condiţional este folosit ca o prescurtare pentru instrucţiuni simple de tipul `if-else`. Forma sa generală este:

```
exprTest ? expresieDa : expresieNu;
```

Mai întâi se evaluează `exprTest` urmată fie de evaluarea lui `expresieDa` fie de cea a lui `expresieNu`, rezultând astfel valoarea întregii expresii. `expresieDa` este evaluată dacă `exprTest` are valoarea `true`; în caz contrar se evaluează `expresieNu`. Prioritatea operatorului condiţional este chiar deasupra operatorilor de atribuire. Acest lucru permite omiterea parantezelor atunci când asignăm rezultatul operatorului condiţional unei variabile. Ca un exemplu, minimul a două variabile poate fi calculat după cum urmează:

```
valMin = x < y ? x : y;
```

2.6 Metode

Ceea ce în alte limbaje de programare numeam procedură sau funcţie, în Java este numit *metodă*. O definiţie completă a noţiunii de metodă o vom da mai târziu, când vom introduce noţiunile de clasă şi obiect. În acest paragraf prezentăm doar câteva noţiuni elementare pentru a putea scrie funcţii de genul celor din C sau Pascal pe care să le folosim în câteva programe simple.

Digitized by Google

Listing 2.4: Declararea şi apelul unei metode

```
public class Minim
{
  public static void main( String[] args )
  {
    int a = 3 ;
    int b = 7 ;
    System.out.println( "Minimul este: " + min(a,b) ) ;
  }

  //declaratia metodei min
  public static int min( int x, int y )
  {
    return x<y?x:y ;
  }
}
```

Antetul unei metode constă dintr-un *nume*, o *listă* (eventual vidă) *de parametri* şi un *tip* pentru valoarea returnată. Codul efectiv al metodei, numit adeseori corpul metodei, este format dintr-o instrucţiune compusă (o secvenţă de instrucţiuni cuprinsă între acolade). Definirea unei metode constă în *antet* şi *corp*. Un exemplu de definire şi utilizare a unei metode este dat în programul din **Listing 2.4**.

Prin prefixarea metodelor cu ajutorul cuvintelor cheie `public static` putem mima într-o oarecare măsură funcţiile din Pascal şi C. Deşi această tehnică este utilă în anumite situaţii, ea nu trebuie utilizată în mod abuziv.

Numele metodei este un identificator. *Lista de parametri* constă din 0 sau mai mulţi *parametri formali*, fiecare având un tip precizat. Când o metodă este apelată, *parametrii actuali* sunt trecuţi în parametrii formali utilizând atribuirea obişnuită. Aceasta înseamnă că tipurile primitive sunt transmise utilizând exclusiv transmiterea prin valoare. Parametrii actuali nu vor putea fi modificaţi de către funcţie. Definirile metodelor pot apărea în orice ordine.

Instrucţiunea `return` este utilizată pentru a întoarce o valoare către codul apelant. Dacă tipul funcţiei este `void` atunci nu se întoarce nici o valoare. În anumite situaţii, pentru ieşirea forţată dintr-o metodă care are tipul `void` se foloseşte `return;` fără nici un parametru.

2.6.1 Supraîncărcarea numelor la metode

Să presupunem că dorim să scriem o metodă care calculează maximul a trei numere întregi. Un antet pentru această metodă ar fi:

Digitized by Google

```
int max(int a, int b, int c)
```

În unele limbaje de programare (Pascal, C), acest lucru nu ar fi permis dacă există deja o funcţie `max` cu doi parametri. De exemplu, se poate să avem deja declarată o metodă `max` cu antetul:

```
int max(int a, int b)
```

Java permite *supraîncărcarea* (engl. *overloading*) numelui metodelor. Aceasta înseamnă că mai multe metode cu acelaşi nume pot fi declarate în cadrul aceleiaşi clase atâta timp cât s*emnăturile* lor (adică lista de parametri) diferă. Atunci când se face un apel al metodei `max`, compilatorul poate uşor să deducă despre care metodă este vorba examinând *lista parametrilor de apel.* Se poate să existe metode supraîncărcate cu acelaşi număr de parametri formali, atâta timp cât cel puţin unul din tipurile din lista de parametri este diferit.

De reţinut faptul că tipul funcţiei nu face parte din semnătura ei. Aceasta înseamnă că nu putem avea două metode în cadrul aceleiaşi clase care să difere doar prin tipul valorii returnate. Metode din clase diferite pot avea acelaşi nume, parametri şi chiar tip returnat, dar despre aceasta vom discuta pe larg mai târziu.

Rezumat

În acest capitol am discutat despre noţiunile fundamentale ale limbajului Java, cum ar fi tipurile primitive, operatorii, instrucţiunile conditionale şi repetitive, precum şi metode, care se regăsesc în aproape orice limbaj de programare.

Totuşi, orice program nebanal presupune utilizarea tipurilor neprimitive, numite *tipuri referinţă*, care vor fi discutate în capitolul următor.

Noţiuni fundamentale

`break`: instrucţiune prin care se iese din cel mai din interior ciclu iterativ sau din cadrul instrucţiunii `switch`.

comentarii: au rolul de a face codul mai lizibil pentru programatori, dar nu au nici o valoare semantică. Java dispune de trei tipuri de comentarii.

`continue`: instrucţiune prin care se trece la următoarea iteraţie din cadrul ciclului iterativ în care este folosită.

`do`: instrucţiune repetitivă în care ciclul repetitiv se execută cel puţin o dată.

`for`: instrucţiune repetitivă utilizată mai ales pentru iteraţie.

identificator: nume dat unei variabile sau unei metode.

`if`: instrucţiune de decizie.

Digitized by Google

instrucţiune compusă: mai multe instrucţiuni cuprinse între acolade şi care formează astfel o singură instucţiune.

`main`: metodă specială în cadrul unei clase, care este apelată la execuţia aplicaţiei.

metodă: echivalentul Java al unei funcţii.

metodă de tip `static`: metodă echivalentă cu o metodă globală.

operator condiţional (? :): operator folosit ca o prescurtare a secvenţelor simple `if...else`.

operatori aritmetici binari: utilizaţi pentru operaţii aritmetice de bază, cum ar fi +, −, *, / şi %.

operatori de atribuire: operatori utilizaţi pentru modificarea valorii unei variabile. Exemple de operatori de atribuire sunt =, +=, −=, *=, /=.

operatori de egalitate: sunt == şi !=. Ei întorc fie `true` fie `false`.

operatori de incrementare şi decrementare: operatori care adaugă, respectiv scad, o unitate la/din valoarea curentă a variabilei. Există două forme incrementare şi decrementare: prefixat şi postfixat.

operatori logici: `&&`, `||` şi `!`, folosiţi pentru a reprezenta conceptele de disjuncţie, conjuncţie şi negare din algebra booleană.

operatori relaţionali: <, >, <=, >= sunt folosiţi pentru a decide care din două valori este mai mică sau mai mare; rezultatul lor este `true` sau `false`.

`return`: instrucţiune utilizată pentru a returna informaţii către instrucţiunea care a apelat metoda.

semnătura unei metode: este formată din combinaţia dintre numele metodei şi tipurile din lista de parametri. Tipul valorii returnate nu face parte din semnătură.

`switch`: instrucţiune `if` generalizată.

tipuri întregi: `byte`, `short`, `int` şi `long`.

tipuri primitive: formate din tipurile întregi, virgulă mobilă, boolean şi caracter.

`while`: instrucţiune iterativă.

Erori frecvente

1. Adăugarea neintenţionată a simbolului `;` imediat după instrucţiunile `for`, `while` conduce la erori logice greu de detectat, deoarece instrucţiunile care se presupune că ar face parte din ciclurile respective, sunt de fapt în afara ciclului, motiv pentru care se vor executa o singură dată. În realitate, `;` reprezintă instrucţiunea vidă, şi este tratată de compilator ca orice altă instrucţiune.

Digitized by Google

2. O metodă care în antet declară că trebuie să returneze o valoare, dar în corpul de definire nu face acest lucru, conduce la apariţia unei erori de compilare.

3. Pentru operaţii logice ŞI/SAU se folosesc `&&` şi `||`, şi nu operatorii `&` şi `|`.

4. Adăugarea unui 0 în faţa unui număr, transformă numărul respectiv în baza 8. De exemplu, dacă numărului 37, îi adăugăm un 0, obţinând 037, acesta devine reprezentarea în baza 8 a unui alt număr, 31.

5. Pentru compararea valorilor stocate în variabile, folosiţi == şi nu =. De exemplu, `if (a == 0)` este corect, în timp ce `if (a = 0)` nu este corect şi compilatorul va semnala o eroare (în loc de `boolean` a găsit `int`).

6. Nu stocaţi într-o variabilă o valoare mai mare/mică decât limitele permise. De exemplu, dacă într-o variabilă de tip `byte`, care poate stoca valori între -128 şi 127, se încearcă stocarea valorii 128, atunci vom avea o *depăşire* (overflow), situaţie în care numărul trece de la o extremă la alta, adică -128, şi va continua numărătoarea de acolo. Situaţiile de depăşire a intervalului admis sunt greu de depistat într-un program, de aceea este indicat să vă alegeţi un tip de variabilă care să ofere spaţiul necesar de stocare (de exemplu, `int` sau `long`).

7. Clauza `else` aparţine de cea mai apropiată instrucţiune `if`. Dacă dorim să asociem clauza `else` cu o instrucţiune if deschisă anterior, trebuie instrucţiunile din cadrul acelui `if` să fie cuprinse între acolade, ca în exemplul de mai jos:

```
if(a>0)
{
 if(b>0)
    c = a+b ;
}
else //else va fi asociat cu if(a>0)
 c = a*b ;
```

8. Numele clasei Java trebuie să fie acelaşi cu numele fişierului sursă care o conţine; atenţie la litere mari şi mici!

9. Lipsa instrucţiunii `break` pe ramurile logice ale instrucţiunii `switch` duce la apariţia unui efect nedorit în cele mai multe cazuri: de la ramura

Digitized by Google

curentă se trece la cea următoare care este şi ea executată. Ca o consecinţă se execută mai multe ramuri din cadrul instrucţiunii `switch`.

Exerciţii

Pe scurt

1. Ce extensii folosesc fişierele sursă şi cele compilate în Java?
2. Descrieţi cele trei tipuri de comentarii în Java.
3. Care sunt cele opt tipuri de date primitive în Java?
4. Care este diferenţa dintre operatorii `*` şi `*=` ?
5. Explicaţi diferenţa dintre operatorii unari prefixaţi şi cei postfixaţi.
6. Descrieţi cele trei tipuri de instrucţiuni de ciclare în Java.
7. Descrieţi toate modurile de utilizare ale instrucţiunii `break`. Ce înseamnă o instrucţiune `break` etichetată?
8. Ce face instrucţiunea `continue`?
9. Ce înseamnă supraîncărcarea de metode?
10. Ce înseamnă transmiterea de parametri prin valoare?

Teorie

1. Fie `b` cu valoarea `5` şi `c` cu valoarea `8`. Care sunt valorile lui `a`, `b` şi `c` după execuţia fiecărei linii de cod de mai jos?

```
a = b++ + c++ ;
a = b++ + ++c ;
a = ++b + c++ ;
a = ++b + ++c ;
```

2. Care este rezultatul expresiei `true && false || true` ?
3. Daţi un exemplu pentru care ciclul `for` de mai jos *nu* este echivalent cu ciclul `while` de mai jos:

Digitized by Google

```
for (init; test; actualizare)
{
    instructiuni;
}
```

```
init;
while (test)
{
    instructiuni;
    actualizare;
}
```

Indicaţie
În cazul în care ciclul `for` conţine o instucţiune `continue`, înainte de a trece la următoarea iteraţie se va executa `actualizare`.

4. Pentru programul de mai jos, care sunt valorile posibile la afişare?

```
public class WhatIsX
{
    public static void f(int x)
    {
        /* corpul functiei este necunoscut */
    }

    public static void main(String[] args)
    {
        int x = 0;
        f(x);
        System.out.println(x);
    }
}
```

În practică

1. Scrieţi o instrucţiune `while` echivalentă cu ciclul `for` de mai jos. La ce ar putea fi utilizat un astfel de ciclu?

```
for( ; ; )
    instructiune;
```

2. Scrieţi un program care afişează numerele de la 1 la 100. Modificaţi apoi programul pentru a întrerupe ciclul de afişare după numărul 36 folosind instrucţiunea `break`. Încercaţi apoi să folosiţi `return` în loc de `break`. Care este diferenţa?

Digitized by Google

3. Scrieţi un program care generează tabelele pentru înmulţirea şi adunarea numerelor cu o singură cifră.

4. Scrieţi un program care să genereze 25 de valori aleatoare de tip `int`. Pentru fiecare valoare generată folosiţi o instrucţiune `if-else` pentru a determina dacă este mai mică, egală sau mai mare decât o altă valoare generată aleator.

5. Scrieţi un program care afişează toate numerele prime cu valori cuprinse între 0 şi Integer.MAX_INT. Folosiţi două cicluri for (unul pentru fiecare număr şi unul pentru testarea divizibilităţii) şi operatorul %.

6. Scrieţi două metode statice. Prima să returneze maximul a trei numere întregi, iar a doua maximul a patru numere întregi.

7. Scrieţi o metodă statică care primeşte ca parametru un an şi returnează `true` dacă anul este bisect şi `false` în caz contrar.

Digitized by Google

3. Referinţe

Noi nu reţinem zile; reţinem doar momente.

Autor anonim

În capitolul 2 am prezentat tipurile primitive din Java. Toate tipurile care nu fac parte dintre cele opt tipuri primitive, inclusiv tipuri importante cum ar fi stringuri, şiruri şi fişiere, sunt tipuri referinţă.

În acest capitol vom învăţa:

- Ce este un tip referinţă şi ce este o variabilă referinţă;
- Prin ce diferă un tip referinţă de un tip primitiv;
- Exemple de tipuri referinţă, incluzând stringuri şi şiruri.

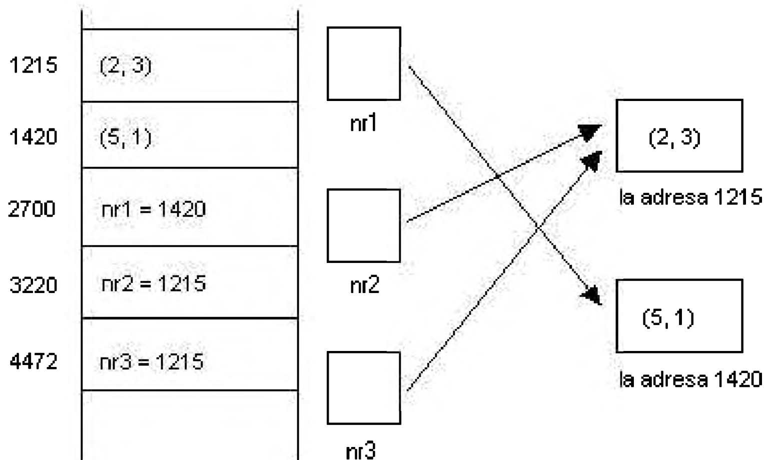
3.1 Ce este o referinţă?

În capitolul 2 am examinat cele opt tipuri primitive împreună cu câteva operaţii care pot fi realizate pe variabile având aceste tipuri. Toate celelalte tipuri de date din Java sunt referinţe. Ce este deci o referinţă?

> O *variabilă referinţă* în Java (numită adeseori simplu *referinţă*) este o variabilă care reţine adresa de memorie la care se află un anumit obiect.

Digitized by Google

Figura 3.1: Ilustrarea unei referinţe: Obiectul de tip Complex stocat la adresa de memorie 1215 este referit atât de către nr2 cât şi de către nr3. Obiectul de tip Complex stocat la adresa 1420 este referit de către nr1. Locaţiile de memorie unde sunt reţinute variabilele au fost alese arbitrar.



Ca un exemplu, în **Figura 3.1** există două obiecte de tipul `Complex`[1]. Presupunem că aceste obiecte au fost stocate la adresele de memorie 1215 şi respectiv 1420. Pentru aceste două obiecte am definit trei referinţe, `nr1`, `nr2` şi `nr3`. Atât `nr2` cât şi `nr3` referă (indică) obiectul stocat la adresa 1215; `nr1` referă obiectul stocat la adresa 1420. Aceasta înseamnă că atât `nr2` cât şi `nr3` au valoarea 1215, iar `nr1` va avea valoarea 1420. Reţineţi că locaţiile efective, cum ar fi 1215 şi 1420, sunt atribuite de compilator la discreţia sa (unde găseşte memorie liberă). În consecinţă, aceste valori nu sunt utile efectiv ca valori numerice. Totuşi, faptul că `nr2` şi `nr3` au *aceeaşi* valoare *este* folositor: înseamnă că ele referă acelaşi obiect.

O referinţă stochează întotdeauna adresa la care un anumit obiect se află, cu excepţia situaţiei când nu referă nici un obiect. În acest caz va stoca *referinţa nulă*, notată în Java cu `null`. Limbajul Java nu permite referinţe către tipurile primitive (cum ar fi `int` sau `float`).

Există două categorii distincte de operaţii care se pot aplica variabilelor referinţă:

[1] Care conţin partea reală şi cea imaginară a unui număr complex.

Digitized by Google

1. Prima categorie permite examinarea şi manipularea valorii referinţă. De exemplu, dacă modificăm valoarea stocată în `nr1` (care este 1420), putem să facem ca `nr1` să refere un alt obiect. Putem de asemenea compara `nr1` şi `nr3` pentru a vedea dacă referă acelaşi obiect;

2. A doua categorie de operaţii se aplică obiectului care este referit. Am putea de exemplu examina sau modifica starea unuia dintre obiectele de tipul `Complex` (am putea examina partea reală şi cea imaginară a unui obiect de tipul `Complex`). Accesul la oricare obiect în Java se face exclusiv prin intermediul unei referinţe către acel obiect.

Înainte de a descrie ce se poate face cu ajutorul referinţelor, să descriem ceea ce nu se poate face. Să considerăm expresia `nr1 * nr2`. Deoarece valorile reţinute de `nr1` şi `nr2` sunt respectiv 1420 şi 1215, produsul lor ar fi 1725300. Totuşi acest calcul este complet lipsit de sens şi rezultatul său nu are nici o valoarea practică. Variabilele referinţă reţin adrese şi nu poate fi asociată nici o semnificaţie logică înmulţirii adreselor.

Analog, `nr1++` nu are nici un sens în Java; ar sugera ca `nr1` - care are valoarea 1420 - să fie crescut la 1421, dar în acest caz nu ar mai referi un obiect valid. Multe alte limbaje de programare, cum ar fi C, definesc noţiunea de *pointer* care are un comportament similar cu cel al unei variabile referinţă. Totuşi, pointerii în C sunt mult mai periculoşi, deoarece este permisă aritmetica pe adresele stocate. În plus, deoarece C permite pointeri şi către tipurile primitive trebuie avut grijă pentru a distinge între aritmetica pe adrese şi aritmetica pe variabilele care sunt referite. Acest lucru se face prin *dereferenţierea* explicită a pointerului. În practică, pointerii limbajului C tind să provoace numeroase erori greu detectabile, care pot adeseori provoca dureri de cap până şi programatorilor experimentaţi!

În Java, singurele operaţii care sunt permise asupra referinţelor (cu o singură excepţie pentru Stringuri), sunt atribuirea prin intermediul operatorului = şi comparaţia prin intermediul operatorilor == şi != . De exemplu, prin atribuirea lui `nr3` a valorii lui `nr1`, vom face ca `nr3` să refere acelaşi obiect pe care îl referă `nr1`. Acum expresia `nr1 == nr3` este adevărată, deoarece ambele referinţe stochează valoarea 1420 şi referă deci acelaşi obiect. `nr3 != nr2` este de asemenea adevărată, deoarece `nr3` şi `nr2` referă acum obiecte distincte.

Cealaltă categorie de operaţii se referă la obiectul care este referit. Există doar trei acţiuni fundamentale care pot fi realizate:

1. Aplicarea unei conversii de tip;

2. Accesul la un câmp al obiectului sau apelul unei metode prin operatorul punct (.);

Digitized by Google

3. Utilizarea operatorului `instanceof` pentru a verifica dacă obiectul reţinut are un anumit tip.

Secţiunea următoare ilustrează mai detaliat operaţiile pe referinţe.

3.2 Fundamente despre obiecte şi referinţe

În Java un *obiect* este orice variabilă care nu este de tip primitiv. Obiectele sunt tratate diferit faţă de tipurile primitive. Variabilele de tipuri primitive sunt manipulate prin valoare, ceea ce înseamnă că valorile lor sunt reţinute în acele variabile şi sunt copiate dintr-o variabilă primitivă în altă variabilă primitivă în timpul instrucţiunii de atribuire. După cum am arătat în secţiunea anterioară, variabilele referinţă stochează referinţe către obiecte.

Obiectul în sine este stocat undeva în memorie, iar variabila referinţă stochează adresa de memorie a obiectului. Astfel, variabila referinţă nu este decât un *nume* pentru acea zonă de memorie. În consecinţă variabilele primitive şi cele referinţă vor avea un comportament diferit. Prezenta secţiune examinează mai în detaliu aceste diferenţe şi ilustrează operaţiile care sunt permise asupra tipurilor referinţă.

3.2.1 Operatorul punct (.)

Operatorul punct (.) este folosit pentru a selecta o metodă care se aplică unui obiect. De exemplu, să presupunem că avem un obiect de tip `Cerc` care defineşte metoda `arie`. Dacă variabila `unCerc` este o referinţă către un obiect de tip `Cerc`, atunci putem calcula aria cercului referit (şi salva această arie într-o variabilă de tip `double`) astfel:

```
double arieCerc = unCerc.arie();
```

Este posibil ca variabila `unCerc` să reţină referinţa `null`. În acest caz, aplicarea operatorului punct va genera o excepţie `NullPointerException` (prezentarea excepţiilor este realizată în capitolul 6) la execuţia programului. De obicei această excepţie va determina terminarea anormală a programului.

Operatorul punct poate fi folosit şi pentru a accesa componentele individuale ale unui obiect, dacă cel care a proiectat obiectul permite acest lucru. Capitolul următor descrie cum se poate face acest lucru şi tot acolo vom explica de ce în general este preferabil să nu se permită accesul direct la componentele individuale ale unui obiect.

Digitized by Google

3.2.2 Declararea obiectelor

Am văzut deja care este sintaxa pentru declararea variabilelor primitive. Pentru obiecte există o diferenţă importantă. Atunci când declarăm o referinţă, nu facem decât să furnizăm un nume care poate fi utilizat pentru a referi un obiect stocat în prealabil în memorie. Totuşi, declaraţia în sine *nu* furnizează şi acel obiect. Să presupunem, de exemplu, că avem un obiect de tip `Cerc` căruia dorim să îi calculăm aria folosind metoda `arie`. Să considerăm secvenţa de instrucţiuni de mai jos:

```
Cerc unCerc; //unCerc poate referi un obiect de tip Cerc
double arieCerc = unCerc.arie(); //calcul arie pentru cerc
```

Totul pare în regulă cu aceste instrucţiuni, până când ne aducem aminte că `unCerc` este numele unui obiect oarecare de tip `Cerc`, dar nu am creat nici un cerc efectiv. În consecinţă, după ce se declară variabila `unCerc`, aceasta va conţine valoarea `null`, ceea ce înseamnă că `unCerc` încă nu referă un obiect `Cerc` valid. Rezultă deci că a doua linie din secvenţa de cod de mai sus este incorectă, deoarece încercăm să calculăm aria unui cerc care încă nu există (este ca şi când am încerca să ne suim la volanul unei maşini pe care încă nu o avem). În exemplul de faţă chiar compilatorul va detecta eroarea, afirmând că `unCerc` "nu este iniţializat". În alte situaţii mai complexe compilatorul nu va putea detecta eroarea şi se va genera o eroare `NullPointerException` doar la execuţia programului.

Singura posibilitate (normală) de a aloca memorie unui obiect Java este folosirea cuvântului cheie `new`. `new` este folosit pentru a construi un nou obiect. Astfel, secvenţa de cod de mai sus corectată este:

```
Cerc unCerc;    //unCerc poate referi un obiect de tip Cerc
unCerc = new Cerc(); //acum unCerc refera un obiect alocat
double arieCerc = unCerc.arie(); //calcul arie pentru cerc
```

Remarcaţi parantezele care se pun după numele obiectului.

Adeseori programatorii combină declararea şi iniţializarea obiectului ca în exemplul de mai jos:

```
Cerc unCerc = new Cerc(); //acum unCerc refera un obiect alocat
double arieCerc = unCerc.arie(); //calcul arie pentru cerc
```

Multe obiecte pot fi de asemenea construite cu anumite valori iniţiale. De exemplu, obiectul de tip `Cerc` ar putea fi construit cu trei parametri, doi pentru coordonatele centrului şi unul pentru lungimea razei.

```
//cerc cu centru (0, 0) si de raza 10
Cerc unCerc = new Cerc(0, 0, 10);
//calcul ariei pentru cerc
double arieCerc = unCerc.arie();
```


Digitized by Google

3.2.3 Colectarea de gunoaie (garbage collection)

Deoarece toate obiectele trebuie construite, ne-am putea aştepta ca atunci când nu mai este nevoie de ele să trebuiască să le distrugem. Totuşi, în Java, când un obiect din memorie nu mai este referit de nici o variabilă, memoria pe care o consumă va fi eliberată automat. Această tehnică se numeşte *colectare de gunoaie.*

3.2.4 Semnificaţia operatorului =

Să presupunem că avem două variabile de tipuri primitive, `x` şi `y`. În această situaţie, semnificaţia instrucţiunii de atribuire

```
x = y;
```

este simplă: valoarea stocată în `y` este transferată în variabila primitivă `x`. Modificările ulterioare ale lui `y` nu afectează în nici un fel valoarea lui `x`.

Pentru referinţe, semnificaţia lui = este *exact* aceeaşi: se copiază valorile referinţelor, adică adresele pe care acestea le indică. Dacă `x` şi `y` sunt referinţe (de tipuri compatibile), atunci, după operaţia de atribuire, `x` va referi acelaşi obiect ca şi `y`. Ceea ce se copiază în acest caz sunt adrese. Obiectul pe care `x` îl referea înainte nu mai este referit de `x`. Dacă `x` a fost singura referinţă către acel obiect, atunci obiectul nu mai este referit acum de nici o variabilă şi este disponibil pentru colectarea de gunoaie. Reţineţi faptul că obiectele nu se copiază prin operatorul =.

Iată câteva exemple. Să presupunem că dorim să creăm două obiecte de tip `Cerc` pentru a calcula suma ariilor lor. Creăm mai întâi obiectul `cerc1`, după care încercăm să creăm obiectul `cerc2` prin modificarea lui `cerc1` după cum urmează (vezi şi **Figura 3.2**):

```
Cerc cerc1 = new Cerc(0, 0, 10); //un cerc de raza 10
Cerc cerc2 = cerc1;
cerc2.setRaza(20); // modificam raza la 20
double arieCercuri = cerc1.arie() + cerc2.arie(); //calcul arie
```

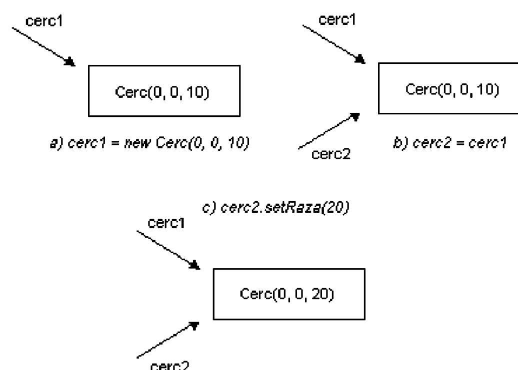
Acest cod nu va funcţiona corect, deoarece nu s-a construit decât un singur obiect de tip `Cerc`. Astfel, cea de-a doua instrucţiune nu face decât să spună că `cerc2` este un alt nume pentru `cerc1`, construit anterior. Cercul construit în prima linie are acum două nume. A treia instrucţiune modifică raza cercului la 20, dar de fapt se modifică raza unicului cerc creat, deci ultima linie adună aria aceluiaşi cerc de rază 20.

Secvenţa de cod corectă este:

```
Cerc cerc1 = new Cerc(0, 0, 10); //un cerc de raza 10
```


Digitized by Google

Figura 3.2: `cerc1` şi `cerc2` indică acelaşi obiect. Modificarea razei lui `cerc2` implică şi modificarea razei lui `cerc1`.




```
Cerc cerc2 = new Cerc();
cerc2.setRaza(20); // modificam raza la 20
double arieCercuri = cerc1.arie() + cerc2.arie(); //calcul arie
```

La o primă vedere, faptul că obiectele nu pot fi copiate, pare să fie o limitare severă. În realitate însă nu este deloc aşa, deşi ne trebuie un pic de timp pentru a ne obişnui cu acest lucru. Există totuşi anumite situaţii când trebuie să copiem obiecte; în aceste situaţii se va folosi metoda `clone()`. `clone()` foloseşte `new()` pentru a crea un nou obiect duplicat. În această carte metoda `clone()` nu este folosită.

3.2.5 Transmiterea de parametri

Din cauza faptului că apelul se face prin valoare, parametrii actuali (de apel) se transpun în parametri formali folosind atribuirea obişnuită. Dacă parametrul trimis este un tip referinţă, atunci ştim deja că prin atribuire atât parametrul formal, cât şi parametrul de apel vor referi acelaşi obiect. Orice metodă aplicată parametrului formal este astfel implicit aplicată şi parametrului de apel. În alte limbaje de programare acest tip de apel se numeşte *apelare prin referinţă*. Utilizarea acestei noţiuni în Java ar fi oarecum nepotrivită, deoarece ne-ar putea face să credem că transmiterea referinţelor s-ar face în mod diferit de tipurile primitive. În realitate, transmiterea parametrilor nu s-a modificat; ceea ce s-a modificat sunt *parametrii* în sine, care nu mai sunt tipuri primitive, ci tipuri referinţă. Metoda `f` de mai jos primeşte ca parametri un obiect de tip `Cerc` şi o valoare întreagă.

Digitized by Google

```
public void f(Cerc unCerc, int val)
{
  unCerc.setRaza(val); //modifica raza obiectului apelant
  val += 5; //nu are nici un efect asupra parametrului actual
}
```

Secvenţa de cod

```
int v = 5;
Cerc cerc = new Cerc();
f(cerc, v);
```

va avea ca efect modificarea razei obiectului `cerc` la 5, dar valoarea lui `v` va rămâne nemodificată. Explicaţie acestui fenomen este simplă: `cerc` şi `unCerc` sunt variabile referinţă *distincte* care indică *acelaşi* obiect, în timp ce `v` şi `val` sunt variabile de tip primitiv *distincte*, şi în consecinţă modificarea uneia nu are nici un efect asupra celeilalte. Pentru a fi mai clar, ne putem imagina că la apelul metodei `f()`, au loc atribuirile

```
unCerc = cerc;
v = val;
```

după care se execută secvenţa de instrucţiuni a metodei.

3.2.6 Semnificaţia operatorului ==

Pentru tipurile primitive operaţia == are valoarea `true` dacă acestea au valori identice. Pentru tipuri referinţă semnificaţia lui == este diferită, dar perfect consistentă cu discuţia din paragraful anterior.

Două variabile referinţă sunt egale via == dacă ele referă acelaşi obiect (s-au ambele sunt `null`). Să considerăm următorul exemplu:

```
Cerc cerc1 = new Cerc(0, 0, 10); //un cerc de raza 10
Cerc cerc2 = new Cerc(0, 0, 10); //un alt cerc tot de raza 10
Cerc cerc3 = cerc2;
```

În acest caz avem două obiecte. Primul este cunoscut sub numele de `cerc1`, al doilea este cunoscut sub două nume: `cerc2` şi `cerc3`. Expresia `cerc2 == cerc3` este adevărată. Totuşi, deşi `cerc1` şi `cerc2` referă obiecte care au valori egale, expresia `cerc1 == cerc2` este falsă. Chiar dacă cele două obiecte au valori egale (ambele sunt cercuri cu centrul în origine şi rază de 10), ele sunt entităţi distincte. Aceleaşi reguli se aplică şi pentru operatorul != .

Cum facem însă pentru a vedea dacă obiectele referite sunt identice? De exemplu, cum putem să verificăm faptul că `cerc1` şi `cerc2` referă obiecte `Cerc` care sunt identice? Obiectele pot fi comparate folosind metoda `equals`. Vom vedea în curând un exemplu de folosire a lui `equals`, în momentul în

Digitized by Google

care vom discuta despre tipul `String`. Fiecare obiect are o metodă `equals`, care, în mod implicit, nu face altceva decât testul == descris mai sus. Pentru ca `equals` să funcţioneze corect, programatorul trebuie să redefinească această metodă pentru obiectele pe care le crează (redefinirea metodelor va fi prezentată în capitolul următor).

3.2.7 Supraîncărcarea operatorilor pentru obiecte

În afara unei singure excepţii pe care o vom discuta în paragraful următor (concatenarea de şiruri), operatorii nu pot fi definiţi pentru a lucra cu obiecte[2]. Astfel, nu există operatorul < pentru nici un fel de obiect. Pentru acest scop, va trebui definită o metodă, cum ar fi `lessThan`, care va realiza comparaţia.

3.3 Şiruri de caractere (stringuri)

Şirurile de caractere în Java sunt definite folosind tipul `String`. Limbajul Java face să pară că `String` este un tip primitiv, deoarece pentru el sunt definiţi operatorii + şi += pentru concatenare (am arătat în paragraful anterior că operatorii nu pot fi aplicaţi obiectelor). Totuşi, acesta este singurul tip referinţă pentru care Java a permis supraîncărcarea operatorilor (pentru comoditatea scrierii programelor). În rest, `String` se comportă ca orice alt obiect.

3.3.1 Fundamentele utilizării stringurilor

Există două reguli fundamentale referitoare la obiectele de tip `String`. Prima este aceea că, exceptând operatorul de concatenare, obiectele `String` se comportă ca toate celelalte obiecte. A doua regulă este aceea că stringurile sunt ne-modificabile. Aceasta înseamnă că, odată construit, un obiect de tip `String` nu mai poate fi modificat.

Deoarece obiectele de tip `String` nu se pot modifica, putem folosi liniştiţi operatorul = pentru ele, fără nici un risc. Iată un exemplu:

```
String vid = "";
String mesaj = "Salutare!";
String repetat = mesaj;
```

După aceste declaraţii, există două obiecte de tip `String`. Primul este un şir vid şi este referit de variabila `vid`, iar al doilea este şirul `"Salutare!"`,

[2] Aceasta este o diferenţă notabilă între Java şi C++, care permite supraîncărcarea operatorilor pentru obiecte. Inginerii de la Sun au considerat că supraîncărcarea operatorilor pentru obiecte aduce mai multe probleme decât beneficii şi au decis ca Java să nu permită acest lucru.

Digitized by Google

care este referit de variabilele `mesaj` şi `repetat`. Pentru majoritatea obiectelor, faptul că obiectul este referit de două variabile ar putea genera probleme. Totuşi, deoarece stringurile nu pot fi modificate, partajarea lor nu pune nici un fel de probleme. Singura posibilitate de a modifica valoarea către care referă variabila `repetat` este aceea de a construi un nou obiect de tip `String` şi a-l atribui lui `repetat`. Această operaţie nu va avea nici un efect asupra valorii pe care o referă `mesaj`.

3.3.2 Concatenarea stringurilor

Java nu permite supraîncărcarea operatorilor pentru tipurile referinţă. Totuşi, pentru comoditate, se acordă o excepţie specială pentru concatenarea obiectelor de tipul `String`.

Atunci când cel puţin unul dintre operanzi este de tip `String`, operatorul + realizează concatenarea. Rezultatul este o referinţă către un obiect nou construit de tip `String`. Iată câteva exemple:

```
"Sunt" + " curajos!"  //-> "Sunt curajos!"
2 + " mere"           //-> "2 mere", 2 este convertit la String
"mere " + 2           //-> "mere 2"
"a" + "b" + "c"       //-> "abc"
```

Şirurile de caractere formate dintr-un singur caracter NU trebuie înlocuite cu constante de tip caracter (constantele caracter sunt de fapt numere).

Java dispune şi de operatorul += pentru şiruri de caractere. Efectul instrucţiunii `str += expr` este `str = str + expr`. Cu alte cuvinte `str` va referi un nou `String` generat de `str + expr`.

Iată un exemplu:

```
String s1 = "ab";
String s2 = "cd";
s1 += s2; //s1 va deveni "abcd"
System.out.println(s1); //va afisa "abcd"
```

Este important să observăm că între atribuirea:

```
i = i + 5  //i este un intreg
```

şi atribuirea:

```
str = str + "hello"  //str este un String
```

există o diferenţă esenţială. În primul caz, variabila `i` este incrementată cu 5; locaţia de memorie a lui `i` nu se modifică. În al doilea caz, se crează un nou string având valoarea `str + "hello"`. După atribuire, `str` va referi acest nou string. Fostul string referit va fi supus colectării de gunoaie (*garbage-collection*) dacă nu a existat o altă referinţă către el.

Digitized by Google

3.3.3 Compararea stringurilor

Deoarece operatorul de adunare funcţionează pe şiruri de caractere, am fi tentaţi să credem că funcţionează şi operatorii relaţionali. Acest lucru nu este însă adevărat.

Conform regulii privind supraîncărcarea operatorilor, operatorii relaţionali (<, <=, >, >=) nu sunt definiţi pentru obiecte de tip `String`. Mai mult, operatorii == şi != au semnificaţia clasică pentru obiecte de tip referinţă (compară adrese şi nu obiecte). De exemplu, pentru două obiecte de tip `String`, `x` şi `y`, expresia `x == y` este adevărată doar dacă `x` şi `y` referă acelaşi obiect de tip `String`. Astfel, dacă `x` şi `y` referă obiecte diferite cu conţinut identic, expresia `x == y` este falsă. Acelaşi raţionament este valabil şi pentru !=.

Pentru a testa egalitatea a două obiecte de tip `String`, se foloseşte metoda `equals`. Expresia `x.equals(y)` este adevărată dacă şirurile de caractere referite de `x` şi de `y` sunt identice.

Ca exemplu să considerăm următoarele şiruri:

```
String x = "ab";
String y = "ab";
String z = y;
```

În această situaţie, expresia `x==y` va fi falsă pentru că `x` şi `y` sunt referinţe către două obiecte diferite (chiar dacă se întâmplă ca acestea să aibe acelaşi conţinut, şi anume `"ab"`), în timp ce expresia `y == z` va fi adevărată pentru că `y` şi `z` sunt referinţe către acelaşi obiect. Pentru a compara cele două şiruri de caractere referite de `x` şi `y`, se foloseşte metoda `equals()`. În concluzie, `x.equals(y)` va fi o expresie adevărată.

Un test mai general poate fi realizat cu metoda `compareTo()`. Utilizând expresia `x.compareTo(y)`, se compară două obiecte de tip `String`, `x` şi `y`. Valoarea returnată este un număr negativ, zero sau un număr pozitiv dacă `x` este mai mic, egal, respectiv mai mare decât `y` din punct de vedere al ordinii lexicografice.

Compararea stringurilor poate fi realizată şi cu ajutorul unei alte metode, numită `compareToIgnoreCase()`. Această metodă are acelaşi comportament cu `compareTo()`, doar că metoda `compareToIgnoreCase()` nu face distincţie între literele mici şi literele mari ale alfabetului.

De exemplu, fie:

```
String s1 = "AbcD";
String s2 = "abcd";
```

În acest caz, expresia `s1.compareTo(s2)` va fi falsă, pentru că `"A"` este diferit de `"a"`, în timp ce expresia `s1.compareToIgnoreCase(s2)` va fi adevărată pentru că metoda `compareToIgnoreCase` nu face diferenţa între

Digitized by Google

literele mari şi cele mici. Altfel spus, din punctul de vedere al metodei `compareToIgnoreCase`, `"A"` este egal cu `"a"`. Este evident că metoda `compareToIgnoreCase` nu are sens să fie folosită pe şiruri de caractere care nu conţin litere, ca în exemplul următor `s1.compareToIgnoreCase("123")`.

3.3.4 Alte metode pentru stringuri

Lungimea unui obiect de tip `String` (un şir vid are lungimea 0) poate fi obţinută cu metoda `length()`. Deoarece `length()` este o metodă, în momentul apelului parantezele sunt necesare.

Există două metode pentru a accesa caracterele din interiorul unui `String`. Metoda `charAt` returnează caracterul aflat la poziţia specificată (primul caracter este pe poziţia 0). Metoda `substring` returnează o referinţă către un `String` nou construit. Metoda are ca parametri poziţia de început şi poziţia primului caracter neinclus.

Iată un exemplu de folosire a acestor metode:

```
String mesaj = "Hello";
int lungimeMesaj = mesaj.length();          //lungimea este 5
char ch = mesaj.charAt(1);                  //ch este 'e'
String subSir = mesaj.substring(2, 4);      //sub este "ll"
```

Limbajul Java oferă o paletă impresionantă de metode pentru manipularea şirurilor de caractere. Pe lângă cele prezentate anterior, metode des utilizate sunt de asemenea:

- `concat()`

  Descriere: metoda `concat` adaugă la sfârşitul unui şir de caractere un alt şir de caractere. Deoarece un obiect de tip `String` nu este modificabil (mutabil), prin concatenare se va crea un nou obiect, ce va fi referit de variabila referinţă `s1`. Fostul obiect referit de `s1` va fi supus colectării de gunoaie, în situaţia în care nici o altă variabilă nu îl referă.

  Antet: `String concat(String str)`

  Exemplu:

  ```
  String s1 = "a";
  s1.concat("b"); //s1 devine "ab"
  ```

- `endsWith()`

  Descriere: verifică dacă şirul de caractere are ca sufix un alt şir de caractere.

Digitized by Google

Antet: `boolean endsWith(String sufix)`

Exemplu:

```
String s1 = "abcde";

System.out.println(s1.endsWith("de"));
/*
 * va afisa true, pentru ca sirul de caractere "abcde" se
 * termina cu sirul de caractere "de"
 */
```

- `equalsIgnoreCase()`

  Descriere: verifică dacă două şiruri sunt egale, fără a face diferenţa între literele mari şi cele mici ale alfabetului .

  Antet: `boolean equalsIgnoreCase(String altString)`

  Exemplu:

```
String s1 = "ABc";

System.out.println(s1.equalsIgnoreCase("abc"));
/*
 * va afisa true, pentru ca nu se face diferenta intre
 * literele mici si cele mari ale alfabetului
 */
```

- `getBytes()`

  Descriere: converteşte stringul într-un şir de `bytes`(octeţi), astfel încât fiecărui caracter din şir îi va corespunde un întreg de tip `byte`, reprezentând primul byte din codul Unicode al caracterului respectiv[3].

  Antet: `byte[] getBytes()`

  Exemplu:

```
byte[] b1 = "abcd".getBytes();
/*
 * sirul b1 va contine codurile Unicode ale caracterelor
 * ce formeaza sirul "abcd", adica 97, 98, 99, 100
 */
```

[3] În realitate, procesul este ceva mai complex: metoda `getBytes()` va transforma codul Unicode al fiecărui caracter din şir (având 2 octeţi) în codul pe 1 octet specific platformei pe care rulează maşina virtuală.

Digitized by Google

- `indexOf()`

  Descriere: returnează poziţia la care se află prima apariţie a unui subşir într-un şir. Dacă subşirul nu se regăseşte în şirul respectiv, metoda returnează valoarea -1.

  Antet: `int indexOf(String str)`

  Exemplu:

```
String s1 = "Limbajul Java este orientat pe obiecte";
String s2 = "Java";
String s3 = "C++";

System.out.println(s1.indexOf(s2));
// va afisa 9 (indicele din s1 la care incepe subsirul s2)

System.out.println(s1.indexOf(s3));
// va afisa -1 (sirul s1 nu contine subsirul s3)
```

- `lastIndexOf()`

  Descriere: returnează poziţia la care se află ultima apariţie a unui subşir într-un şir. Dacă subşirul nu se află în şirul respectiv, metoda returnează valoarea -1.

  Antet: `int lastIndexOf(String str)`

  Exemplu:

```
String s1 = "El invata Java pentru ca Java e OOP";
String s2 = "Java";

System.out.println(s1.lastIndexOf(s2));
/*
 * va afisa 25 (indicele la care incepe ultima aparitie
 * a subsirului s2 in sirul s1)
 */
```

- `replace()`

  Descriere: înlocuieşte apariţiile unui caracter într-un şir cu un alt caracter şi returnează noul şir astfel format. Dacă respectivul caracter nu apare în şir, atunci noul şir va fi la fel cu cel iniţial.

  Antet: `String replace(char carVechi, char carNou)`

  Exemplu:

```
String s1 = "Java";
String s2 = s1.replace('a', 'i');
// s2 va avea valoarea "Jivi"
```


Digitized by Google

- `startsWith()`

  Descriere: verifică dacă un şir de caractere are ca prefix un alt şir de caractere.

  Antet: `boolean startsWith(String prefix)`

  Exemplu:

```
String s1 = "abcd";
System.out.println(s1.startsWith("abc"));
/*
 * va afisa true pentru ca stringul s1 incepe
 * cu sirul de caractere "abc"
 */
```

- `toLowerCase()`

  Descriere: converteşte toate literele mari din string în litere mici.

  Antet: `String toLowerCase()`

  Exemplu:

```
String s1 = "ABCdE";
String s2 = s1.toLowerCase();
//s2 va fi egal cu "abcde"
```

- `toUpperCase()`

  Descriere: este opusa metodei `toLowerCase`. Converteşte toate literele mici în litere mari.

  Antet: `String toUpperCase()`

  Exemplu:

```
String s1 = "abCdE";
String s2 = s1.toUpperCase();
//s2 va fi egal cu "ABDCE"
```

- `trim()`

  Descriere: elimină spaţiile de la începutul şi sfârşitul unui şir de caractere.

  Antet: `String trim()`

  Exemplu:

```
String s1 = "    Java          ";
s1 = s1.trim();
//s1 va fi egal cu "Java"
```


Digitized by Google

3.3.5 Conversia de la string la tipurile primitive şi invers

Metoda toString() poate fi utilizată pentru a converti orice tip primitiv la String. De exemplu, toString(45) returnează o referinţă către şirul nou construit "45". Majoritatea obiectelor furnizează o implementare a metodei toString(). De fapt, atunci când operatorul + are un operand de tip String, operandul care nu este de tip String este automat convertit la String folosind metoda toString(). Pentru tipurile de date numerice, există o variantă a metodei toString() care permite precizarea unei anumite baze. Astfel, instrucţiunea:

```
System.out.println(Integer.toString(55, 2));
```

are ca efect tipărirea reprezentării în baza 2 a numărului 55.

Pentru a converti un String la un int există posibilitatea de a folosi metoda Integer.parseInt(). Această metodă generează o excepţie dacă String-ul convertit nu conţine o valoare întreagă. Pentru a obţine un double dintr-un String se poate utiliza metoda parseDouble(). Iată două exemple:

```
int x = Integer.parseInt("75");
double y = Double.parseDouble("3.14");
```

3.4 Şiruri

Şirurile[4] sunt structura fundamentală prin care se pot reţine mai multe elemente de acelaşi tip. În Java, şirurile nu sunt tipuri primitive; ele se comportă foarte asemănător cu un obiect. Din acest motiv, multe dintre regulile care sunt valabile pentru obiecte se aplică şi la şiruri.

Fiecare element dintr-un şir poate fi accesat prin mecanismul de *indiciere* oferit de operatorul [ ]. Spre deosebire de limbajele C sau C++, Java verifică validitatea indicilor[5].

În Java, ca şi în C, şirurile sunt întotdeauna indiciate de la 0. Astfel, un şir a cu 3 elemente este format din a[0], a[1], a[2]. Numărul de elemente care pot fi stocate în şirul a este permanent reţinut în variabila a.length. Observaţi că aici (spre deosebire de String-uri) nu se pun paranteze. O parcurgere tipică a unui şir este:

[4]Desemnate în alte lucrări şi sub numele de *tablouri, vectori, blocuri*.

[5]Acesta este un lucru foarte important care vine în ajutorul programatorilor, mai ales a celor începători. Indicii a căror valoare depăşeşte numărul de elemente alocat, sunt adeseori cauza multor erori obscure în C şi C++. În Java, accesarea unui şir cu un indice în afara limitei este imediat semnalată prin excepţia ArrayIndexOutOfBoundsException.

Digitized by Google

```
for (int i = 0; i < a.length; i++)
```

3.4.1 Declaraţie, atribuire şi metode

Un şir de elemente întregi se declară astfel:

```
int[] sir1;
```

Deoarece un şir este un obiect, declaraţia de mai sus nu alocă memorie pentru şir. Variabila `sir1` este doar un nume (referinţă) pentru un şir de numere întregi, şi în acest moment valoarea ei este `null`. Pentru a aloca 100 de numere întregi, vom folosi instrucţiunea:

```
sir1 = new int[100];
```

Acum `sir1` este o referinţă către un şir de 100 de numere întregi.

Există şi alte posibilităţi de a declara şiruri. De exemplu, putem unifica declaraţia şi alocarea de memorie într-o singură instrucţiune:

```
int[] sir1 = new int[100];
```

Se pot folosi şi liste de iniţializare, ca în C sau C++. În exemplul următor se alocă un şir cu patru elemente, care va fi referit de către variabila `sir2`:

```
int[] sir2 = {3, 4, 6, 19};
```

Parantezele pătrate de la declarare pot fi puse fie înainte, fie după numele şirului. Plasarea parantezelor înainte de nume face mai vizibil faptul că este vorba de un şir, de aceea vom folosi această notaţie.

Declararea unui şir de obiecte (deci nu tipuri primitive) foloseşte aceeaşi sintaxă. Trebuie să reţineţi însă că după alocarea şirului, fiecare element din şir va avea valoarea `null`. Pentru fiecare element din şir trebuie alocată memorie separat. De exemplu, un şir cu 5 cercuri se construieşte astfel:

```
//declaram un sir de cercuri
Cerc[] sirDeCercuri;

//alocam memorie pentru 5 referinte la Cerc
sirDeCercuri = new Cerc[5];

for (int i = 0; i < sirDeCercuri.length; ++i)
{
    sirDeCercuri[i] = new Cerc();
    //se aloca un obiect Cerc referintei nr. i
}
```

Programul din **Listing 3.1** ilustrează modul de folosire al şirurilor în Java. În jocul de loterie se selectează săptămânal şase numere de la 1 la 49. Programul

Digitized by Google

alege aleator numere pentru 1000 de jocuri şi afişează apoi de câte ori a apărut fiecare număr în cele 1000 de jocuri. Linia 14 declară un şir de numere întregi care reţine de câte ori a fost extras fiecare număr. Deoarece indicierea şirurilor începe de la 0, adunarea cu 1 este esenţială. Fără această adunare am fi avut un şir cu elemente de la 0 la 48, şi orice acces la elementul cu indicele 49 ar fi generat o excepţie `ArrayIndexOutOfBoundsException` (este adevărat că procedând în acest mod, elementul de pe poziţia 0 a rămas neutilizat). Ciclul din liniile 15-18 iniţializează valorile şirului cu 0. Restul programului este relativ simplu. Se foloseşte din nou metoda `Math.random()` care generează un număr în intervalul `[0,1)`. Rezultatele sunt afişate în liniile 28-31.

Listing 3.1: Program demonstrativ pentru şiruri

```
1 //clasa demonstrativa pentru siruri
2 public class Loterie
3 {
4   //genereaza numere de loterie intre 1 si 49
5   //afiseaza numarul de aparitii al fiecarui numar
6   //declaratii constante:
7   public static final int NUMERE = 49;
8   public static final int NUMERE_PE_JOC = 6;
9   public static final int JOCURI = 1000;
10  //main
11  public static final void main(String[] args)
12  {
13    //genereaza numerele
14    int[] numere = new int[NUMERE + 1];
15    for (int i = 0; i < numere.length; ++i)
16    {
17      numere[i] = 0;
18    }
19
20    for (int i = 0; i < JOCURI; ++i)
21    {
22      for (int j = 0 ; j < NUMERE_PE_JOC; ++j)
23      {
24        numere[(int) (Math.random() * 49) + 1]++ ;
25      }
26    }
27    //afisare rezultate
28    for (int k = 1; k <= NUMERE; ++k)
29    {
30      System.out.println(k + ": " + numere[k]);
31    }
32  }
33 }
```


Digitized by Google

Dat fiind faptul că şirul este un tip referinţă, operatorul = nu copiază şiruri. De aceea dacă x şi y sunt şiruri, efectul secvenţei de instrucţiuni:

```
int[] x = new int[100];
int[] y = new int[100];
...
x = y;
```

este că x şi y referă acum al doilea şir.

Şirurile pot fi utilizate ca parametri pentru metode. Regulile de transmitere se deduc logic din faptul că şirul este o referinţă. Să presupunem că avem o metodă f care acceptă un şir de int ca parametru. Apelul şi definirea arată astfel:

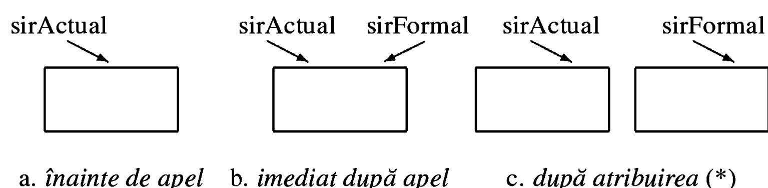
```
f(sirActual);                    //apelul metodei
void f(int[] sirFormal);         //declaratia metodei
```

Conform convenţiilor de transmitere a parametrilor în Java pentru tipurile referinţă, variabilele sirActual şi sirFormal referă acelaşi obiect (la transmiterea parametrului se copiază *valoarea* lui sirActual, care este o simplă adresă). Astfel, accesul la elementul sirFormal[i] este de fapt un acces la elementul sirActual[i]. Aceasta înseamnă că variabilele conţinute în şir pot fi modificate de către metodă. O observaţie importantă este aceea că linia de cod din cadrul metodei f:

```
sirFormal = new int [20]; // (*)
```

nu are nici un efect asupra lui sirActual. Acest lucru se datorează faptului că în Java transmiterea parametrilor se face prin valoare, iar la revenirea din funcţie adresa pe care o referă sirActual rămâne nemodificată. Instrucţiunea de mai sus nu face decât să schimbe şirul către care referă sirFormal (vezi figura următoare).

Figura 3.3: Transmiterea parametrilor în Java



Digitized by Google

Deoarece numele şirurilor sunt doar nişte referinţe, o funcţie poate să returneze un şir.

3.4.2 Expansiunea dinamică a şirurilor

Să presupunem că dorim să citim o secvenţă de stringuri şi să o reţinem într-un şir. Una dintre proprietăţile fundamentale ale şirurilor este aceea că înainte de a fi utilizate, trebuie să alocăm memorie pentru un număr fix de elemente care vor fi stocate. Dacă nu ştim de la bun început câte elemente vor fi stocate în şir, va fi dificil să alegem o valoare rezonabilă pentru dimensiunea şirului. Această secţiune prezintă o metodă prin care putem extinde dinamic şirul dacă dimensiunea iniţială se dovedeşte a fi prea mică. Această tehnică poartă numele de *expansiune dinamică a şirurilor* şi permite alocarea de şiruri de dimensiune arbitrară pe care le putem redimensiona pe măsură ce programul rulează.

Alocarea obişnuită de memorie pentru şiruri se realizează astfel:

```
String[] a = new String[10];
```

Să presupunem că după ce am făcut această declaraţie, la un moment dat avem nevoie de 12 elemente şi nu de 10. În această situaţie putem folosi următoarea manevră:

```
String [] original = a;          //salvam referinta lui a
a = new String[12];              //alocam din nou memorie
for (int i = 0; i < 10; i++)     //copiem elementele in a
{
    a[i] = original[i];
}
```

Un moment de gândire este suficient pentru a ne convinge că această operaţie este consumatoare de resurse, deoarece şirul original trebuie copiat înapoi în noul şir `a`. De exemplu, dacă extensia dinamică a numărului de elemente ar trebui făcută ca răspuns la citirea de date, ar fi ineficient să expansionăm ori de câte ori citim câteva elemente. Din acest motiv, de câte ori se realizează o extensie dinamică, numărul de elemente este crescut cu un coeficient *multiplicativ*. Am putea de exemplu dubla numărul de elemente la fiecare expansiune dinamică. Astfel, dintr-un şir cu `N` elemente, generăm un şir cu `2N` elemente, iar costul expansiunii este împărţit între cele `N` elemente care pot fi inserate în şir fără a realiza extensia.

Pentru ca lucrurile să fie concrete, **Listing 3.2** prezintă un program care citeşte un număr nelimitat de stringuri de la tastatură şi le reţine într-un şir a cărui dimensiune este extinsă dinamic. Oprirea din citire se realizează în momentul în care ultimul string introdus este egal cu stringul `"end"`. Funcţia

Digitized by Google

`resize()` realizează expansiunea (sau contracţia!) şirului returnând o referinţă către un şir nou construit. Similar, metoda `getStrings()` returnează o referinţă către şirul în care sunt citite elementele.

La începutul lui `getStrings()`, `nrElemente` este iniţializat cu 0 şi alocăm memorie pentru 5 elemente. În linia 34 citim în mod repetat câte un element. Dacă şirul este "umplut", lucru indicat de intrarea în testul de la linia 36, atunci şirul este expansionat prin apelul metodei `resize()`. Liniile 64-74 realizează expansiunea şirului folosind strategia prezentată anterior. La linia 42, elementul citit este stocat în tablou, iar numărul de elemente citite este incrementat. În final, în linia 51 contractăm şirul la numărul de elemente citite efectiv. Linia 1 conţine o noţiune nouă, directiva `import`, care va fi prezentată pe larg în secţiunea 4.4. De asemenea liniile 31 şi 45 conţin instrucţiunea `try-catch`, care va fi prezentată în cadrul capitolului 6.

Listing 3.2: Program pentru citirea unui număr nelimitat de stringuri urmată de afişarea lor

```
1 import java.io.* ;
2 /** Citirea unui numar nelimitat de stringuri. */
3 public class ReadStrings
4 {
5   public static void main(String[] args)
6   {
7     String[] array = getStrings();
8     printSingleDimensionalArray(array);
9
10    char[][] characters = stringsToChars(array);
11    printMultiDimensionalArray(characters);
12  }
13
14  /**
15   * Citeste un numar nelimitat de stringuri
16   * pana intalneste stringul "end",
17   * fara a trata erorile
18   */
19  public static String[] getStrings()
20  {
21    //BufferedReader este prezentata in cap. Java I/O
22    BufferedReader in = new BufferedReader(
23    new InputStreamReader(System.in));
24
25    String[] elemente = new String[5]; //se aloca 5 elemente
26    int nrElemente = 0; //numarul de elemente citite
27    String s; //sir in care se citeste cate o linie
28
29    System.out.println("Introduceti stringuri:");
30
```


Digitized by Google

```
    try
    {
      //cat timp linia este diferita de "end"
      while (!"end".equalsIgnoreCase(s = in.readLine()))
      {
        if (nrElemente == elemente.length)
        {
          //dubleaza dimensiunea sirului "umplut"
          elemente = resize(elemente,
          elemente.length * 2);
        }
        elemente[nrElemente++] = s;
      }
    }
    catch (Exception e)
    {
      //nu se trateaza exceptia
    }
    System.out.println("Citire incheiata.");

    return resize(elemente, nrElemente);
    //trunchiaza sirul la numarul de elemente citite
  }

  /** Afiseaza sirul de stringuri citit. */
  public static void printSingleDimensionalArray(String[] array)
  {
    System.out.println("Elementele citite sunt:");
    for (int i = 0; i < array.length; i++)
      System.out.println(array[i]);
  }

  /** Redimensioneaza sirul. */
  public static String[] resize(String[] sir, int dimensiuneNoua)
  {
    int elementeDeCopiat = Math.min(sir.length,
    dimensiuneNoua);
    String[] sirNou = new String[dimensiuneNoua];
    for (int i = 0; i < elementeDeCopiat; ++i)
    {
      sirNou[i] = sir[i];
    }
    return sirNou;
  }

  /** Transforma String[] in char[][]. */
  public static char[][] stringsToChars(String[] array)
  {
    char[][] characters = new char[array.length][5];
    for (int i = 0; i < array.length; i++)
```


Digitized by Google

```
81     {
82       characters[i] = new char[array[i].length()];
83       array[i].getChars(0, characters[i].length, characters[i], 0);
84     }
85
86     return characters;
87   }
88
89   /** Afiseaza o matrice de caractere. */
90   public static void printMultiDimensionalArray(
91           char[][] characters)
92   {
93     System.out.println("Elemente sub forma de caractere:");
94
95     for (int i = 0; i < characters.length; i++)
96     {
97       for (int j = 0; j < characters[i].length; j++)
98         System.out.print(characters[i][j]);
99       System.out.println();
100     }
101   }
102 }
```

3.4.3 Şiruri cu mai multe dimensiuni

În anumite situaţii trebuie să stocăm datele în şiruri cu mai multe dimensiuni. Cel mai des folosite sunt matricele, adică şirurile cu două dimensiuni. Alocarea de memorie pentru şiruri cu mai multe dimensiuni se realizează precizând numărul de elemente pentru fiecare indice, iar indicierea se face plasând fiecare indice între paranteze pătrate. Ca un exemplu, declaraţia:

```
int[][] x = new int[2][3];
```

defineşte matricea `x`, în care primul indice poate fi 0 sau 1, iar al doilea poate lua valori de la 0 la 2.

Listing 3.2 oferă un exemplu simplu de utilizare a unei matrici. După cum se poate observa la linia 10, şirul de stringuri ce a fost citit de la tastatură este transformat într-o matrice de caractere (câte un şir de caractere pentru fiecare string în parte). Metoda `printMultiDimensionalArray` prezintă modul de parcurgere a elementelor matricei obţinute.

3.4.4 Argumente în linie de comandă

Parametrii transmişi în linie de comandă sunt disponibili în cadrul unei aplicaţii, prin examinarea parametrilor funcţiei `main()`. Şirul de stringuri numit

Digitized by Google

`args` din funcţia `main` conţine parametrii transmişi programului Java în linia de comandă. De exemplu, dacă avem un program numit `Suma.java` pe care îl executăm cu comanda:

```
java Suma 2 3
```

atunci parametrul `args[0]` va fi o referinţă către stringul `"2"`, iar parametrul `args[1]` va fi o referinţă către `"3"`. Astfel, programul următor implementează o comandă de adunare a numerelor trimise ca argument:

Listing 3.3: Utilizarea argumentelor în linie de comandă

```
public class Suma
{
  //afiseaza suma parametrilor primiti in linie de comanda
  public static void main(String[] args)
  {
    if (args.length == 0)
    {
      System.out.println("Nu exista argumente");
      return;
    }

    double suma = 0;

    for (int i = 0; i < args.length; ++i)
    {
      suma += Double.parseDouble(args[i]);
    }

    System.out.println("Suma argumentelor: " + suma);
  }
}
```

Rezumat

Capitolul de faţă a prezentat tipurile referinţă. O referinţă este o variabilă care stochează adresa de memorie unde se află un obiect sau referinţa specială `null`. Referinţele pot indica doar obiecte, nu şi variabile de tipuri primitive. Orice obiect poate fi referit prin una sau mai multe variabile referinţă.

Deoarece în Java există doar opt tipuri primitive, aproape totul se traduce în obiecte şi clase. Dintre obiecte, stringurile au un tratament mai special, pentru că se pot folosi pentru concatenare operatorii + şi `+=`. În rest, stringurile sunt la fel ca orice alt tip referinţă. Pentru a testa dacă două stringuri sunt egale ca şi conţinut, se foloseşte metoda `equals`.

Digitized by Google

Şirurile reprezintă o colecţie de elemente de acelaşi tip. Este important să reţinem faptul că indicele de numerotare a elementelor din şir porneşte întotdeauna de la 0.

În cadrul capitolului care urmează vom prezenta modul în care putem defini noi tipuri de date în Java, folosind noţiunea de *clase*.

Noţiuni fundamentale

apelare prin referinţă: în majoritatea limbajelor de programare, aceasta înseamnă că parametrul formal reprezintă o referinţă către parametrul actual. Acest efect este atins în mod natural în Java în momentul în care se utilizează apelul prin valoare pentru tipuri referinţă.

argument în linie de comandă: argumente transmise la execuţia unui program Java şi care sunt preluate în funcţia `main()`.

colectarea de gunoaie: eliberearea automată din memorie a obiectelor care nu mai sunt referite.

construirea: pentru obiecte, se realizează prin intermediul cuvântului cheie `new`.

`equals`: metodă prin care se poate compara dacă valorile stocate de două obiecte sunt egale.

`length` **(atribut)**: folosit pentru a determina dimensiunea unui şir.

`length` **(metodă)**: folosită pentru a determina lungimea unui obiect de tip `String`.

obiect: o entitate de tip neprimitiv.

`new`: folosit pentru a construi noi obiecte.

`null`: referinţă specială prin care se specifică faptul că nu se face referire la nici un obiect.

`NullPointerException`: tip de excepţie generată de încercarea de a accesa un atribut sau o metodă pe o referinţă `null`.

pointer: ca şi referinţa, pointerul conţine adresa la care se află un obiect. Spre deosebire de referinţe, pointerii necesită o dereferenţiere explicită pentru a putea manipula obiectul indicat. În Java nu există pointeri.

referinţă: variabilă care stochează adresa la care se află un obiect, şi prin intermediul căreia poate fi manipulat obiectul care este referit.

şir: reţine o colecţie de obiecte de acelaşi tip

Erori frecvente

1. Pentru tipuri referinţă operatorul = nu copiază valorile obiectelor. El copiază doar adrese.

Digitized by Google

2. Pentru tipuri referinţă (inclusiv stringuri) trebuie folosită metoda `equals` în loc de == pentru a testa dacă obiectele referite sunt egale ca şi conţinut.

3. Alocarea cu un element mai puţin decât trebuie pentru şiruri (indexarea începe de la 0!).

4. Tipurile referinţă sunt implicit iniţializate cu `null`. Nici un obiect nu este construit fără apelul lui `new`. O excepţie `NullPointerException` indică faptul că aţi uitat să alocaţi memorie pentru un obiect.

5. În Java şirurile sunt indexate la 0 la `N - 1`, unde `N` este dimensiunea şirului. Totuşi Java verifică valoarea indicilor, şi accesul în afara limitelor este detectat în timpul execuţiei.

6. Şirurile bidimensionale sunt indexate prin `A[i][j]` şi nu `A[i,j]` ca în Pascal.

7. Folosiţi " " şi nu ' ' pentru a scrie un spaţiu sau alte caractere.

Exerciţii

Pe scurt

1. Care sunt diferenţele majore între tipurile primitive şi tipurile referinţă?

2. Enumeraţi 5 operaţii care se pot aplica unui tip referinţă.

3. Enumeraţi operaţiile de bază care pot fi efectuate pe stringuri.

Teorie

1. Dacă `x` şi `y` au valorile `5` respectiv `7`, care este rezultatul următoarei afişări:

```
System.out.println(x + ' ' + y);
System.out.println(x + " " + y);
```

În practică

1. Creaţi un şir de obiecte de tip `String` şi asociaţi un `String` fiecărui element din şir. Afişaţi apoi elementele şirului în cadrul unui ciclu `for`.

2. Scrieţi o metodă care returnează `true` dacă stringul `str1` este prefix pentru stringul `str2`. Nu folosiţi nici o metodă generală de căutare pe stringuri în afară de `charAt`.

Digitized by Google

3. Scrieţi un program care preia trei argumente de tip String din linia de comandă şi le afişează pe ecran.

Digitized by Google

4. Obiecte şi clase

> Cuvintele sau limbajul, aşa cum sunt ele scrise sau vorbite nu par să joace nici un rol în mecanismul gândirii mele. Obiectele care par să serverască drept elemente în gândirea mea sunt anumite semne şi imagini mai mult sau mai puţin clare care pot fi în mod voluntar reproduse sau combinate.
>
> Albert Einstein

În acest capitol vom începe să discutăm despre programarea orientată pe obiecte (object oriented programming - OOP) în Java. O componentă fundamentală a programării orientate pe obiecte este specificarea, implementarea şi folosirea obiectelor. În capitolul anterior am văzut deja câteva exemple de obiecte, cum ar fi stringurile, care fac parte din bibliotecile limbajului Java. Am putut observa şi faptul că fiecare obiect este caracterizat de o anumită stare care poate fi modificată prin aplicarea operatorului punct (.). În limbajul Java, starea şi funcţionalitatea unui obiect se definesc prin intermediul unei clase. Un obiect este de fapt o instanţă a unei clase.

În cadrul capitolulului de faţă vom prezenta:

- Cum se foloseşte în Java conceptul de clasă pentru a obţine încapsularea şi ascunderea de informaţii, concepte fundamentale ale OOP;
- Cum se implementează o clasă Java;
- Cum se pot grupa clasele în pachete pe baza funcţionalităţii lor comune.

Digitized by Google

4.1 Ce este programarea orientată pe obiecte?

Programarea orientată pe obiecte s-a impus ca modelul dominant al anilor '90 şi continuă să domine şi în deceniul curent. În această secţiune vom prezenta modul în care Java suportă programarea orientată pe obiecte şi vom menţiona câteva dintre principiile ei fundamentale.

În centrul programării orientate pe obiecte se află noţiunea de *obiect*. Obiectul este o variabilă complexă definit de o *structură* şi o *stare*. Fiecare obiect dispune de *operaţii* prin intermediul cărora i se poate manipula starea. Aşa cum am văzut deja, în limbajul Java se face distincţie între un obiect şi o variabilă de un tip primitiv, dar aceasta este o specificitate a limbajului Java şi nu a programării orientate pe obiecte în general. Pe lângă operaţiile cu un caracter general, asupra obiectelor se mai pot realiza şi alte operaţii:

- **Crearea de noi obiecte, însoţită eventual de iniţializarea obiectelor;**
- **Copierea şi testarea egalităţii;**
- **Realizarea de operaţii de intrare/ieşire cu obiecte.**

Obiectul trebuie privit ca o unitate *atomică* pe care utilizatorul nu ar trebui să o disece. În mod normal, nu ne punem problema de a jongla cu biţii din care este format un număr reprezentat în virgulă mobilă şi ar fi de-a dreptul ridicol să încercăm să incrementăm un astfel de număr prin modificarea directă a reprezentării sale interne.

Principiul atomicităţii este cunoscut sub numele de *ascunderea informaţiei*. Utilizatorul nu are acces direct la componentele unui obiect sau la implementarea sa. Acestea vor putea fi accesate doar prin intermediul *metodelor* care au fost furnizate împreună cu obiectul. Putem privi fiecare obiect ca fiind ambalat într-o cutie pe care este scris "Nu deschideţi! Nu conţine componente reparabile de către utilizator!". În viaţa de zi cu zi, majoritatea celor care încearcă să repare componente cu această inscripţie sfârşesc prin a face mai mult rău decât bine. Din acest punct de vedere, programarea *imită* lumea reală. Gruparea datelor şi a operaţiilor asupra acestor date în acelaşi întreg (agregat), având grijă să ascundem detaliile de implementare ale agregatului, este cunoscută sub numele de *încapsulare*. Aşadar, datele sunt ascunse, iar accesul lor se realizează prin intermediul operaţiilor încapsulate împreună cu ele, numite metode.

Unul dintre principalele scopuri ale programării orientate pe obiecte este reutilizarea codului. La fel cum inginerii refolosesc din nou şi din nou aceleaşi componente în proiectarea de componente electronice, programatorii ar trebui

Digitized by Google

să refolosească obiectele în loc să le reimplementeze. Există două situaţii distincte legate de reutilizarea codului (refolosirea obiectelor):

1. Situaţia în care avem deja la dispoziţie un obiect care implementează exact comportamentul pe care îl dorim. Refolosirea obiectului nu pune în acest caz nici un fel de probleme;

2. Situaţia în care dorim să folosim un obiect care deja există, dar care, deşi are un comportament foarte similar cu ceea ce vrem, nu corespunde exact cu necesităţile noastre. Aici este de fapt adevărata provocare: obiectul existent va trebui extins pentru a fi adaptat la comportamentul dorit. Astfel se economiseşte timp de dezvoltare, deoarece adeseori este mult mai uşor să se adapteze un obiect deja existent (a cărui funcţionalitate este asemănătoare cu cea a obiectului de care avem nevoie) decât să se rescrie totul de la zero.

Limbajele de programare orientate pe obiecte furnizează mai multe mecanisme pentru a facilita reutilizarea codului. Unul dintre mecanisme este folosirea codului *generic*. Dacă implementarea este identică, şi diferă doar tipul de bază al obiectului, nu este necesar să rescriem complet codul: vom scrie în schimb un cod generic care funcţionează pentru orice tip. De exemplu, algoritmul de sortare al unui şir de obiecte nu depinde de obiectele care sunt sortate, deci se poate implementa un algoritm generic de sortare.

Moştenirea este un alt mecanism care permite extinderea funcţionalităţii unui obiect. Cu alte cuvinte, putem crea noi tipuri de date care să extindă (sau să restricţioneze) proprietăţile tipului de date original.

Un alt principiu important al programării orientate pe obiecte este *polimorfismul*. Un tip referinţă *polimorfic* poate să refere obiecte de mai multe tipuri. Atunci când se apelează o metodă a tipului *polimorfic*, se va selecta automat metoda care corespunde tipului referit în acel moment (vom descrie în detaliu atât moştenirea cât şi polimorfismul în capitolul următor).

Un *obiect* în Java este o instanţă a unei *clase*. O clasă este similară cu un tip record din Pascal sau cu o structură din C, doar că există două îmbunătăţiri majore. În primul rând, membrii clasei pot fi atât funcţii cât şi date, numite în acest context *metode*, respectiv *atribute*. În al doilea rând, domeniul de vizibilitate al acestor membri poate fi restricţionat. Deoarece metodele care manipulează starea obiectului sunt membri ai clasei, ele sunt accesate prin intermediul operatorului punct, la fel ca şi atributele. În terminologia programării orientate pe obiecte, atunci când apelăm o metodă a obiectului spunem că "trimitem un mesaj" obiectului.

Digitized by Google

4.2 Un exemplu simplu

Să ne amintim că, atunci când proiectăm o clasă, este important să ascundem detaliile interne faţă de utilizatorul clasei. Clasa poate să îşi definească funcţionalitatea prin intermediul *metodelor*. Unele dintre aceste metode vor descrie cum se creează şi se iniţializează o instanţă a clasei, cum se realizează testele de egalitate şi cum se descrie starea clasei. Celelalte metode sunt specifice structurii particulare pe care o are clasa. Ideea este că utilizatorul nu trebuie să aibă dreptul de a modifica direct starea obiectului, ci el va trebui să folosească metodele clasei pentru a realiza acest lucru. Această idee poate fi impusă prin ascunderea anumitor membri faţă de utilizator. Pentru a realiza aceasta, vom preciza ca aceşti membri să fie stocaţi în secţiunea `private`. Compilatorul va avea grijă ca membri din secţiunea `private` să fie inaccesibili utilizatorului acelui obiect. În general, toate atributele unui obiect ar trebui să fie declarate `private`.

Programul din **Listing 4.1** prezintă modul de definire al unei clase care modelează un cerc. Definirea clasei constă în două părţi: `public` şi `private`. Secţiunea `public` reprezintă porţiunea care este vizibilă pentru utilizatorul obiectului. Deoarece datele sunt ascunse faţă de utilizator, secţiunea `public` va conţine în mod normal numai metode şi constante. În exemplul nostru avem două metode, una pentru a scrie şi una pentru a citi raza obiectelor de tip `Circle`. Celelalte două metode calculează aria respectiv lungimea obiectului de tip `Circle`. Secţiunea `private` conţine datele; acestea sunt invizibile pentru utilizatorul obiectului. Atributul `radius` poate fi accesat doar prin intermediul metodelor publice `setRadius()` şi `getRadius()`.

Listing 4.1: Definirea clasei `Circle`

```
/**
* Clasa simpla Java care modeleaza un Cerc.
*/
public class Circle
{
  //raza cercului
  //valoarea razei nu poate fi modificata
  //direct de catre utilizator
  private double radius;

  /** Modifica raza cercului*/
  public void setRadius(double r)
  {
    radius = r;
  }
  /** Metoda ptr a obtine raza cercului.*/
```


Digitized by Google

```
  public double getRadius()
  {
    return radius;
  }
  /** Metoda ptr calculul ariei cercului.*/
  public double area()
  {
    return Math.PI * radius * radius;
  }
  /** Metoda ptr calculul lungimii.*/
  public double length()
  {
    return 2 * Math.PI * radius;
  }
}
```

Programul din **Listing 4.2** prezintă modul de folosire al unui obiect de tip `Circle`. Deoarece `setRadius()`, `getRadius()`, `area()` şi `length()` sunt membri ai clasei, ei sunt accesaţi folosind operatorul punct. Atributul `radius` ar fi putut şi el să fie accesat folosind operatorul punct, dacă nu ar fi fost declarat de tip `private`. Accesarea lui `radius` din linia 15 ar fi fost ilegală dacă nu era comentată folosind `//`.

Să rezumăm terminologia învăţată. O clasă conţine *membri* care pot fi *atribute* (câmpuri, date) sau *metode* (funcţii). Metodele pot acţiona asupra atributelor şi pot apela alte metode. Modificatorul de vizibilitate `public` face ca membrul respectiv să fie accesibil oricui prin intermediul operatorului punct. Modificatorul de vizibilitate `private` face ca membrul respectiv să fie accesibil doar metodelor clasei. Dacă nu se pune nici un modificator de vizibilitate, atunci accesul la membru este de tip *friendly*, despre care vom vorbi în paragraful 4.4.4. Mai există şi un al patrulea modificator, numit `protected` pe care îl vom prezenta în capitolul următor.

Listing 4.2: Testarea clasei `Circle`

```
1  //clasa simpla de testare a clasei Circle
2  public class TestCircle
3  {
4    public static void main(String[] args)
5    {
6      Circle circle = new Circle();
7
8      circle.setRadius(10);
9      System.out.println("Raza=" + circle.getRadius());
10     System.out.println("Aria=" + circle.area());
11     System.out.println("Lungimea=" + circle.length());
12
13     //urmatoarea linie ar genera o
```


Digitized by Google

```
    //eroare de compilare
    //circle.radius = 20;
  }
}
```

4.3 Metode uzuale

Metodele unei clase se pot grupa în două categorii:

- metode clasice, uzuale, care se regăsesc în (aproape) toate clasele;
- metode care definesc comportamente specifice unei anumite clase.

În această secţiune vom prezenta prima categorie de metode: constructorii, modificatorii, accesorii, `toString()` şi `equals()`.

4.3.1 Constructori

Aşa cum am menţionat deja, una dintre proprietăţile fundamentale ale obiectelor este că acestea pot fi definite şi, eventual, iniţializate. În limbajul Java, metoda care controlează modul în care un obiect este creat şi iniţializat este *constructorul*. Deoarece Java permite supraîncărcarea metodelor, o clasă poate să definească mai mulţi constructori.

Dacă la definirea clasei nu se furnizează nici un constructor, cum este cazul clasei `Circle` din **Listing 4.1**, compilatorul creează automat un constructor implicit care iniţializează fiecare membru cu valorile implicite. Aceasta înseamnă că atributele de tipuri primitive sunt iniţializate cu 0 (cele boolean cu `false`), iar atributele de tip referinţă sunt iniţializate cu `null`. Astfel, în cazul nostru, atributul `radius` va avea implicit valoarea 0.

Pentru a furniza un constructor, vom scrie o metodă care are acelaşi nume cu clasa şi care nu returnează nimic. În **Listing 4.3** avem doi constructori: unul începe la linia 7, iar celălalt la linia 15. Folosind aceşti doi constructori vom putea crea obiecte de tip `Date` în următoarele moduri:

```
Date d1 = new Date();
Date d2 = new Date(15, 3, 2000);
```

De remarcat faptul că odată ce aţi definit un constructor pentru o clasă, compilatorul nu mai generează constructorul implicit fără parametri. Dacă veţi avea nevoie de un constructor fără parametri, va trebui acum să îl scrieţi singuri. De exemplu, constructorul din linia 7 trebuie definit obligatoriu pentru a putea crea un obiect de tipul celui referit de către `d1`.

Digitized by Google

Listing 4.3: O clasă `Date` minimală care ilustrează constructorii şi metodele `equals()` şi `toString()`

```
//clasa Java simpla pentru stocarea unei
//date calendaristice
//nu se face validarea datelor
public class Date
{
  //constructor fara parametri
  public Date()
  {
    day = 1;
    month = 1;
    year = 2000;
  }

  //constructor cu trei parametri
  public Date(int theDay, int theMonth, int theYear)
  {
    day = theDay;
    month = theMonth;
    year = theYear;
  }

  //test de egalitate
  //intoarce true daca Obiectul x
  //este egal cu obiectul curent
  public boolean equals(Object x)
  {
    if (!(x instanceof Date))
      return false;
    Date date = (Date) x;
    return date.day == day && date.month == month
      && date.year == year;
  }

  //conversie la String
  public String toString()
  {
    return day + "/" + month + "/" + year ;
  }

  //atribute
  private int day;
  private int month;
  private int year;
}
```


Digitized by Google

4.3.2 Modificatori şi accesori

Atributele sunt declarate de obicei ca fiind `private`. Aceasta înseamnă că ele nu vor putea fi direct accesate de către rutinele care nu aparţin clasei. Există totuşi multe situaţii în care dorim să examinăm sau chiar să modificăm valoarea unui atribut.

O posibilitate este aceea de a declara atributele ca fiind `public`. Aceasta nu este o soluţie elegantă, deoarece încalcă principiul ascunderii informaţiei. Putem însă scrie metode care să examineze sau să modifice valoarea fiecărui câmp. O metodă care citeşte, dar nu modifică starea unui obiect este numită *accesor*. O metodă care modifică starea unui obiect este numită *modificator* (engl. mutator).

Cazuri particulare de accesori şi modificatori sunt cele care acţionează asupra unui singur câmp. Accesorii de acest tip au un nume care începe de obicei cu `get`, cum ar fi `getRadius()`, iar modificatorii au un nume care începe de regulă cu `set`, cum ar fi `setRadius()`. Aceste metode sunt atât de des folosite în Java, încât au şi denumiri consacrate în limbajul uzual al programatorilor Java: *getteri*, respectiv *setteri* (a nu se confunda cu rasa de patrupede care poartă acelaşi nume).

Avantajul folosirii unui modificator (sau a unui setter) este că acesta poate verifica dacă starea obiectului este corectă. Astfel, un setter care modifică atributul `day` al unui obiect de tip `Date` poate verifica corectitudinea datei care rezultă.

4.3.3 Afişare şi `toString()`

În general, afişarea stării unui obiect se face utilizând metoda `print()` din clasa `System.out`. Pentru a putea face acest lucru trebuie ca obiectul care se doreşte a fi afişat să conţină o metodă cu numele de `toString()`. Această metodă întoarce un `String` (reprezentând starea obiectului) care poate fi afişat. Ca un exemplu, în **Listing 4.3** am prezentat o implementare rudimentară a unei metode `toString()` pentru clasa `Date` în liniile 35-38. Definirea acestei metode permite să afişăm un obiect `d1` de tip `Date` folosind instrucţiunea `System.out.print(d1)`. Practic, compilatorul Java nu face altceva decât să invoce automat `toString()` pentru fiecare obiect care se afişează. Astfel `System.out.print(d1)` este tradusă de către compilator în instrucţiunea `System.out.print(d1.toString())`. Această simplă facilitate lasă impresia programatorilor neavizaţi că metoda `print()` "ştie" să afişeze obiecte Java. În realitate, metoda `print()` nu ştie decât să afişeze stringuri. Compilatorul Java este însă suficient de inteligent pentru a transforma obiectele

Digitized by Google

în stringuri, prin apelul metodei `toString()` (de altfel, tot compilatorul este cel care converteşte la afişare şi tipurile primitive la `String`, atunci când este cazul).

4.3.4 Metoda `equals()`

Metoda `equals` este folosită pentru a testa dacă două referinţe indică obiecte care au aceeaşi valoare (stare). Antetul acestei metode este întotdeauna

```
public boolean equals(Object rhs)
```

Aţi remarcat probabil nedumeriţi faptul că parametrul trimis metodei este de tip `Object` şi nu de tip `Date`, cum ar fi fost de aşteptat. Raţiunea acestui lucru o să o prezentăm în capitolul următor. În general, metoda `equals` pentru o clasă `X` este implementată în aşa fel încât să returneze `true` doar dacă `rhs` (abreviere pentru "right hand side", adică operandul din partea dreaptă a expresiei) este o instanţă a lui `X` şi, în plus, după conversia la `X` toate atributele lui `rhs` sunt egale cu atributele obiectulului (atributele de tip primitiv trebuie să fie egale via ==, iar atributele de tip referinţă trebuie să fie egale via `equals()`).

Un exemplu de implementare a lui `equals` este dat în **Listing 4.3** pentru clasa `Date` în liniile 25-32.

4.3.5 Metode statice

Există anumite cuvinte cheie ale limbajului Java care specifică proprietăţi speciale pentru unele *atribute* sau *metode*. Un astfel de cuvânt cheie este `static`. Atributele şi metodele declarate `static` într-o clasă, sunt aceleaşi pentru toate obiectele, adică pentru toate variabilele de tipul acelei clase. Fiind identice pentru toate obiectele unei clase, metodele statice pot fi accesate fără să fie nevoie de o *instanţiere* a clasei respective (adică de o variabilă de clasa respectivă). Metodele statice pot utiliza variabile statice declarate în interiorul clasei.

Cel mai cunoscut exemplu de metodă statică este `main()`. Alte exemple de metode statice pot fi găsite în clasele `String`, `Integer` şi `Math`. Exemple de astfel de metode sunt `String.valueOf()`, `Integer.parseInt()`, `Math.sin()` şi `Math.max()`. Accesul la metodele statice respectă aceleaşi reguli de vizibilitate ca şi metodele normale.

4.3.6 Atribute statice

Atributele statice sunt folosite în situaţia în care avem variabile care trebuie partajate de către toate instanţele unei clase. De obicei atributele statice sunt

Digitized by Google

constante simbolice, dar acest lucru nu este obligatoriu. Atunci când un atribut al unei clase este declarat de tip static, doar o singură instanţă a acelei variabile va fi creată. Ea nu face parte din nici o instanţă a clasei. Ea se comportă ca un fel de variabilă globală unică, vizibilă în cadrul clasei. Cu alte cuvinte, dacă avem declaraţia

```
public class Exemplu
{
    private int x;
    private static int y;
}
```

fiecare obiect de tip `Exemplu` va avea propriul atribut `x`, dar va fi doar un singur `y` partajat de toate instanţele clasei `Exemplu`.

O folosire frecventă pentru câmpurile statice o reprezintă constantele. De exemplu, clasa `Integer` defineşte atributul `MAX_VALUE` astfel

```
public final static int MAX_VALUE = 2147483647;
```

Analog se defineşte şi constanta `PI` din clasa `Math` pe care am folosit-o în clasa `Circle`. Modificatorul `final` indică faptul că identificatorul care urmează este o constantă. Dacă această constantă nu ar fi fost un atribut `static`, atunci fiecare instanţă a clasei `Integer` ar fi avut un atribut cu numele de `MAX_VALUE`, irosind astfel inutil spaţiu în memorie.

Astfel, vom avea o singură variabilă cu numele de `MAX_VALUE`. Constanta poate fi accesată de oricare dintre metodele clasei `Integer` prin identificatorul `MAX_VALUE`. Ea va putea fi folosită şi de către un obiect de tip `Integer` numit `x` prin sintaxa `x.MAX_VALUE`, ca orice alt câmp. Acest lucru este permis doar pentru că `MAX_VALUE` este `public`. În sfârşit, `MAX_VALUE` poate fi folosit şi prin intermediul numelui clasei ca `Integer.MAX_VALUE` (tot pentru că este `public`). Această ultimă folosire nu ar fi fost permisă pentru un câmp care nu este `static`.

Chiar şi fără modificatorul `final`, atributele statice sunt foarte folositoare. Să presupunem că vrem să reţinem numărul de obiecte de tip `Circle` care au fost construite. Pentru aceasta avem nevoie de o variabilă statică. În clasa `Circle` vom face declaraţia:

```
private static int numarInstante = 0;
```

Vom putea apoi incrementa numărul de instanţe în constructor. Dacă acest câmp nu ar fi fost de tip `static` am avea un comportament incorect, deoarece fiecare obiect de tip `Circle` ar fi avut propriul atribut `numarInstante` care ar fi fost incrementat de la 0 la 1.

Remarcaţi faptul că, deoarece un atribut de tip static nu necesită un obiect care să îl controleze, fiind partajat de către toate instanţele clasei, el poate fi

Digitized by Google

folosit de către o metodă statică (dacă regulile de vizibilitate permit acest lucru). Atributele nestatice ale unei clase vor putea fi folosite de către o metodă statică doar dacă se furnizează şi un obiect care să le controleze.

4.3.7 Metoda `main()`

Atunci când este invocat, interpretorul `java` caută metoda `main()` din clasa care i se dă ca parametru. Astfel, în situaţia în care aplicaţia noastră este formată din mai multe clase, putem scrie câte o metodă `main()` pentru fiecare clasă. Acest lucru ne permite să testăm funcţionalitatea de bază a claselor individuale. Trebuie să avem totuşi în vedere faptul că plasarea funcţiei `main()` în cadrul clasei ne conferă mai multă vizibilitate decât ne-ar fi permis prin utilizarea clasei din alt loc. Astfel, apeluri ale metodelor `private` pot fi făcute în test, dar ele vor eşua când vor fi utilizate în afara clasei.

4.4 Pachete

Pachetele sunt folosite pentru a organiza clasele similare. Fiecare pachet constă dintr-o mulţime de clase. Clasele care sunt în acelaşi pachet au restricţii de vizibilitate mai slabe între ele decât clasele din pachete diferite.

Java oferă o serie de pachete predefinite, printre care `java.io`, `java.awt`, `java.lang`, `java.util`, `java.applet` etc. Pachetul `java.lang` include, printre altele, clasele `Integer`, `Math`, `String` şi `System`. Clase mai cunoscute din `java.util` sunt `Date`, `Random`, `StringTokenizer`. Pachetul `java.io` cuprinde diverse clase pentru stream-uri, pe care le vom prezenta în capitolul 7. De asemenea, pentru o prezentare mai amănunţită a pachetelor predefinite, vă recomandăm să consultaţi anexa C. Pe lângă informaţii detaliate despre aceste pachete şi despre modalitatea de a vă crea propriile dumneavoastră pachete, anexa C prezintă şi noţiunea de arhivă *jar*, foarte utilă în cadrul aplicaţiilor de dimensiuni mai mari.

O clasă oarecare `C` dintr-un pachet `P` este desemnată prin `P.C`. De exemplu, putem declara un obiect de tip `Date` care să conţină data şi ora curentă astfel:

```
java.util.Date today = new java.util.Date();
```

Observaţi că prin specificarea numelui pachetului din care face parte clasa, evităm conflictele care pot fi generate de clase cu acelaşi nume din pachete diferite[1].

[1] De exemplu, cu clasa `Date` definită anterior în secţiunea 4.3.

Digitized by Google

4.4.1 Directiva `import`

Utilizarea permanentă a numelui pachetului din care fac parte clasele poate fi uneori deosebit de anevoioasă. Pentru a evita acest lucru se foloseşte directiva `import`:

```
import NumePachet.NumeClasa;
```

sau

```
import NumePachet.*;
```

Dacă recurgem la prima formă a directivei `import`, vom putea folosi denumirea `NumeClasa` ca o prescurtare pentru numele clasei cu calificare completă. Dacă folosim cea de-a doua formă, toate clasele din pachet vor putea fi abreviate cu numele lor neprecedat de numele pachetului.

De exemplu, realizând directivele `import` de mai jos:

```
import java.util.Date;
import java.io.*;
```

putem să folosim:

```
Date today = new Date();
FileReader file = new FileReader(name);
```

Folosirea directivelor `import` economiseşte timpul de scriere. Având în vedere că cea de-a doua formă este mai generală, ea este şi cel mai des folosită. Există două dezavantaje ale folosirii directivei `import`. Primul dezavantaj este acela că la citirea codului va fi mai greu de stabilit pachetul din care o anumită clasă face parte. Al doilea este acela că folosirea celei de-a doua forme poate să introducă prescurtări neintenţionate pentru anumite clase, ceea ce va genera conflicte de denumire care vor trebui rezolvate prin folosirea de nume de clase calificate.

Să presupunem că avem următoarele directive:

```
import java.util.*;  //pachet predefinit
import myutil.*;     //pachet definit de catre noi
```

cu intenţia de a importa clasa `java.util.Random` şi o clasă pe care am definit-o chiar noi. Atunci, dacă noi avem propria clasă `Date` în pachetul `myutil`, directiva `import` va genera un conflict cu `java.util.Date` şi, din această cauză, clasa va trebui să fie complet calificată ori de câte ori este utilizată. Am fi putut evita aceste probleme dacă am fi folosit prima formă

```
import java.util.Random;
```


Digitized by Google

Directivele `import` trebuie să apară înainte de orice declarare a unei clase. Pachetul `java.lang` este automat inclus în întregime. Acesta este motivul pentru care putem folosi prescurtări de genul `Integer.parseInt()`, `System.out`, `Math.max()` etc.

4.4.2 Instrucţiunea `package`

Pentru a indica faptul că o clasă face parte dintr-un anumit pachet trebuie să realizăm două lucruri. În primul rând trebuie să scriem o instrucţiune `package` pe prima linie, înainte de a declara clasa. Apoi, va trebui să plasăm clasa în directorul corespunzător. Mai multe detalii veţi găsi în cadrul secţiunii C.2 a anexei C.

4.4.3 Variabila sistem `CLASSPATH`

Compilatorul şi interpretorul Java caută automat pachetele şi clasele Java în directoarele care sunt precizate în variabila sistem `CLASSPATH`. Ce înseamnă aceasta? Iată o posibilă setare pentru `CLASSPATH`, mai întâi pentru un sistem de operare Windows 95/98/2000, iar apoi pentru un sistem Unix/Linux:

```
SET CLASSPATH=.;c:\jdk1.4\lib\
export CLASSPATH=.:/usr/local/jdk1.4/lib:$HOME/
                                        javawork
```

În ambele cazuri variabila `CLASSPATH` conţine directoarele (sau arhivele) care conţin programele compilate având extensia `.class`. De exemplu, dacă variabila `CLASSPATH` nu este setată corect, nu veţi putea compila nici măcar cel mai banal program Java, deoarece pachetul `java.lang` nu va fi găsit. O clasă care se află în pachetul `P` va trebui să fie pusă într-un director cu numele `P` care să se afle într-un director din `CLASSPATH`. Directorul curent (.) este întotdeauna în variabila `CLASSPATH`, deci dacă lucraţi într-un singur director principal, puteţi crea subdirectoarele chiar în el. Totuşi, în majoritatea situaţiilor, veţi dori să creaţi un subdirector Java separat şi să creaţi directoarele pentru pachete în acel subdirector. În această situaţie va trebui să adăugaţi la variabila `CLASSPATH` acest director. Acest lucru a fost realizat în exemplul de mai sus prin adăugarea directorului `$HOME/javawork` la `CLASSPATH` (notaţia `$HOME`, familiară utilizatorilor de Linux, reprezintă în acest context un alias pentru un director propriu-zis; de exemplu, `/home/george`). Utilizatorii de Windows pot adapta la fel de uşor variabila sistem `CLASSPATH` conform necesităţilor lor. Odată aceste modificări încheiate, în directorul `javawork` veţi putea crea subdirectorul `io`. În subdirectorul `io` vom plasa codul pentru clasa

Digitized by Google

`Reader`. Clasa Reader este o clasă ajutătoare, care ne permite să citim foarte comod, variabile de tip `int`, `char`, `String`, `float` şi chiar şiruri de numere întregi. Exemplele din volumul al doilea al cărţii vor utiliza intensiv această clasă utilitară.

Codul sursă al clasei este prezentat în **Listing** 4.4 (detalii despre instrucţiunea `try-catch` vom prezenta în capitolul 6, iar despre clasele de intrare/ieşire în capitolul 7).

O aplicaţie va putea în această situaţie să folosească metoda `readInt()` fie prin

```
int x = io.Reader.readInt();
```

sau pur şi simplu prin

```
int x = Reader.readInt();
```

dacă se furnizează directiva `import` corespunzătoare.

Listing 4.4: Clasa `Reader`

```
package io;
//clasa va trebui salvata intr-un director cu numele "io"
//directorul in care se afla "io" va trebui adaugat
//in CLASSPATH
import java.io.*;
import java.util.StringTokenizer;
public class Reader
{
  public static String readString()
  {
    BufferedReader in = new BufferedReader(
    new InputStreamReader(System.in));
    try
    {
      return in.readLine();
    }
    catch(IOException e)
    {
      //ignore
    }
    return null;
  }

  public static int readInt()
  {
    return Integer.parseInt(readString());
  }

  public static double readDouble()
  {
```


Digitized by Google

```
    return Double.parseDouble(readString());
  }

  public static char readChar()
  {
    BufferedReader in = new BufferedReader(
    new InputStreamReader(System.in));
    try
    {
      return (char)in.read();
    }
    catch(IOException e)
    {
      //ignore
    }
    return '\0';
  }

  public static int[] readIntArray()
  {
    String s = readString();
    StringTokenizer st = new StringTokenizer(s);
    //aloca memorie pentru sir
    int[] a = new int[st.countTokens()];

    for (int i = 0; i < a.length; ++i)
    {
      a[i] = Integer.parseInt(st.nextToken());
    }

    return a;
  }

}
```

4.4.4 Reguli de vizibilitate package-friendly

Pachetele au câteva reguli de vizibilitate importante. În primul rând, dacă pentru un membru al unei clase nu se precizează nici un modificator de vizibilitate (`public`, `protected` sau `private`), atunci membrul respectiv devine (package) *friendly*. Aceasta înseamnă că acel câmp este vizibil doar pentru clasele din cadrul aceluiaşi pachet. Aceasta este o vizibilitate mai puţin restrictivă decât `private`, dar mai restrictivă decât `public` (care este vizibil şi pentru membrii din alte clase, chiar şi din pachete diferite).

În al doilea rând, doar clasele `public` din cadrul unui pachet pot fi folosite din afara pachetului. Acesta este motivul pentru care am pus întotdeauna modi-

Digitized by Google

ficatorul `public` în faţa unei clase. Clasele nu pot fi declarate `private` sau `protected` (cu o singură excepţie: clasele interioare, despre care vom vorbi în capitolul următor). Accesul de tip *friendly* se extinde şi pentru clase. Dacă o clasă nu este declarată ca fiind de tip `public`, atunci ea va putea fi accesată doar de clasele din cadrul aceluiaşi pachet.

Toate clasele care nu fac parte din nici un pachet, dar sunt accesibile fiind puse într-un director din `CLASSPATH`, sunt considerate automat ca făcând parte din acelaşi pachet implicit. Ca o consecinţă, accesul de tip *friendly* se aplică pentru toate aceste clase. Acesta este motivul pentru care vizibilitatea nu este afectată dacă omitem să punem modificatorul `public` în clasele care nu fac parte dintr-un pachet. Totuşi, această modalitate de folosire a accesului *friendly* nu este recomandată.

4.4.5 Compilarea separată

Atunci când o aplicaţie constă din mai multe fişiere sursă `.java`, fiecare fişier trebuie compilat separat. În mod normal, fiecare clasă este plasată într-un fişier `.java` propriu. Ca urmare a compilării vom obţine o colecţie de fişiere `.class`. Clasele sursă pot fi compilate în orice ordine. Pentru a compila toate fişierele sursă dintr-un director printr-o singură comandă, puteţi folosi o extensie a comenzii `javac`, astfel:

```
javac *.java
```

4.5 Alte operaţii cu obiecte şi clase

În această secţiune vom prezenta încă trei cuvinte cheie importante: `this`, `instanceof` şi `static`. `this` are mai multe utilizări în Java. Două dintre ele le vom prezenta în această secţiune. `instanceof` are şi el mai multe utilizări; îl vom folosi aici pentru a ne asigura că o conversie de tip se poate realiza. Şi cuvântul cheie `static` are mai multe semnificaţii. Vom vorbi despre metode statice, atribute statice şi iniţializatori statici.

4.5.1 Referinţa `this`

Cea mai cunoscută utilizare pentru `this` este ca o referinţă la obiectul curent. Imaginaţi-vă că `this` vă indică în fiecare moment locul unde vă aflaţi. O utilizare tipică pentru `this` este calificarea atributelor unei clase în cadrul

Digitized by Google

unei metode a clasei care primeşte parametri cu nume identic cu numele atributelor. De exemplu, în clasa `Circle`, din **Listing 4.1**, putem defini metoda `setRadius` astfel:

```
/** Modifica raza cercului. */
public void setRadius(double radius)
{
    //radius este parametrul,
    //iar this.radius este atributul clasei
    this.radius = radius;
}
```

În secvenţa de cod precedentă, pentru a face distincţie între parametrul `radius` şi atributul cu acelaşi nume (care este "ascuns" de către parametru) se foloseşte sintaxa `this.radius` pentru a referi atributul clasei.

Un alt exemplu de folosire al lui `this` este pentru a testa că parametrul pe care o metodă îl primeşte nu este chiar obiectul curent. Să presupunem, de exemplu, că avem o clasă `Account` care are o metodă `finalTransfer()` pentru a transfera toată suma de bani dintr-un cont în altul. Metoda ar putea fi scrisă astfel:

```
public void finalTransfer(Account account)
{
    dollars += account.dollars;
    account.dollars = 0;
}
```

Să considerăm secvenţa de cod de mai jos:

```
Account account1;
Account account2;
....
account2 = account1;
account1.finalTransfer(account2);
```

Deoarece transferăm bani în cadrul aceluiaşi cont, nu ar trebui să fie nici o modificare în cadrul contului. Totuşi, ultima linie din `finalTransfer()` are ca efect golirea contului debitor, ceea ce va face ca suma din `account2` să fie nulă. O modalitate de a evita o astfel de situaţie este folosirea unui test pentru pseudonime (referinţe care indică acelaşi obiect):

```
public void finalTransfer(Account account)
{
    //daca se incearca un transfer in acelasi cont
    if (this == account)
    {
        return;
    }

```


Digitized by Google

```
    dollars += account.dollars;
    account.dollars = 0;
}
```

4.5.2 Prescurtarea `this` pentru constructori

Multe clase dispun de mai mulţi constructori care au un comportament similar. Putem folosi `this` în cadrul unui constructor pentru a apela ceilalţi constructori ai clasei. De exemplu, o altă posibilitate de a scrie constructorul fără parametri pentru clasa `Date` este:

```
public Date()
{
    //apeleaza constructorul Date(int, int, int)
    this(1, 1, 2000);
}
```

Se pot realiza şi exemple mai complicate, dar întotdeauna apelul lui `this` trebuie să fie prima instrucţiune din constructor, celelalte instrucţiuni fiind în continuarea acesteia.

4.5.3 Operatorul `instanceof`

Operatorul `instanceof` realizează o testare de tip în timpul execuţiei. Rezultatul expresiei:

```
exp instanceof NumeClasa
```

este `true` dacă `exp` este o instanţă a lui `NumeClasa` şi `false` în caz contrar. Dacă `exp` este `null`, rezultatul este întotdeauna `false`. `instanceof` este folosit de obicei înainte de o conversie de tip, şi adeseori este folosit în legătură cu referinţele polimorfice pe care le vom prezenta în capitolul următor.

4.5.4 Iniţializatori statici

Atributele statice sunt iniţializate atunci când clasa este încărcată în memorie. Uneori este însă nevoie de o iniţializare mai complexă a acestor atribute. Să presupunem de exemplu că avem nevoie de un şir static care specifică pentru fiecare număr între 0 şi 100 dacă este număr prim sau nu. O posibilitate ar fi să definim o metodă statică şi să cerem programatorului să apeleze acea metodă înainte de a folosi şirul.

O alternativă mai elegantă la această soluţie este folosirea *iniţializatorului static*. Instrucţiunile din cadrul iniţializatorului static sunt executate automat la încărcarea clasei în memorie (deci înainte de a fi creată prima instanţă), ceea

Digitized by Google

ce ne asigură că atributele statice sunt corect iniţializate la utilizarea clasei. Un exemplu este prezentat în **Listing 4.5**. Aici iniţializatorul static se extinde de la linia 5 la linia 20. Iniţializatorul static trebuie să urmeze imediat după membrul static.

Listing 4.5: Clasa `Prime`, care foloseşte o secvenţă statică de iniţializare

```
1  public class Prime
2  {
3      private static boolean prime[] = new boolean[100];
4
5      static
6      {
7          for (int i = 2; i < prime.length; ++i)
8          {
9              prime[i] = true;
10             for (int j = 2; j <= i/2; ++j)
11             {
12                 if (i % j == 0)
13                 {
14                     prime[i] = false;
15                     break;
16                 }
17             }
18
19         }
20     }
21     //restul clasei
22 }
```

Rezumat

Acest capitol descrie noţiunile de obiect şi clasă, precum şi elementele fundamentale legate de acestea, cum ar fi atribute, metode, constructori, reguli de vizibilitate şi pachete. Clasa este mecanismul prin care Java permite crearea de noi tipuri referinţă. Pachetele sunt folosite pentru a grupa clase cu comportament similar. Pentru o mai bună înţelegere a pachetelor şi a modului de operare cu ele vă invităm să consultaţi anexa C. Tot acolo veţi afla că toate clasele care compun o aplicaţie pot fi grupate într-un singur fişier, indiferent dacă sunt organizate pe pachete sau nu. Rezultatul va fi o arhivă `jar`.

Capitolul următor prezintă alte concepte importante ale programării orientate pe obiecte: moştenirea, polimorfismul, programarea generică.

Digitized by Google

Noţiuni fundamentale

clasă: concept care grupează atribute şi metode care sunt aplicate instanţelor clasei.

`CLASSPATH`: variabilă sistem care specifică directoarele şi fişierele arhive în care maşina virtuală JVM caută clasele folosite de aplicaţiile Java.

constructor: metodă specială responsabilă cu crearea şi iniţializarea de noi instanţe ale clasei.

constructor `this`: folosit pentru a apela un alt constructor din aceeaşi clasă.

friendly: tip de acces prin care clasele/metodele/atributele asociate nu sunt accesibile decât în interiorul pachetului din care fac parte.

`import`: instrucţiune prin care se oferă o prescurtare pentru un nume complet de clasă (numele complet include şi pachetul din care provine clasa).

iniţializator static: secvenţă din cadrul unei clase care permite iniţializarea atributelor statice.

obiect: instanţă a unei clase.

pachet: termen folosit pentru a organiza colecţii de clase cu comportament similar.

`package`: instrucţiune care indică faptul că o clasă este membră a unui pachet. Trebuie să preceadă definirea unei clase.

`private`: tip de acces prin care atributele/metodele sunt complet ascunse celorlalte clase, putând fi utilizate doar în cadrul clasei din care fac parte.

`public`: tip de acces prin care atributele/metodele unei clase sunt vizibile în restul claselor.

referinţa `this`: referinţă către obiectul curent.

`static`: cuvânt cheie prin care se precizează faptul că o metodă sau un atribut sunt comune pentru toate instanţele clasei. Membrii statici nu necesită o instanţă pentru a fi accesaţi.

Erori frecvente

1. Membrii `private` nu pot fi accesaţi din afara clasei. Reţineţi că, implicit, membrii unei clase sunt `friendly`. Ei sunt vizibili doar în cadrul pachetului.

2. Folosiţi `public class` în loc de `class`, cu excepţia situaţiei în care scrieţi o clasă ajutătoare de care nu veţi mai avea nevoie în afara pachetului.

Digitized by Google

3. Parametrul formal al metodei `equals` trebuie să fie de tip `Object`. În caz contrar, deşi programul se va compila, în anumite situaţii se va apela metoda `equals` implicită (care compară obiectele folosind ==).

4. Metodele statice nu pot accesa atributele nestatice fără un obiect care să le controleze.

5. Deseori se uită adăugarea `".*"` atunci când se doreşte importarea tuturor claselor dintr-un pachet.

6. Clasele care fac parte dintr-un pachet trebuie să se afle într-un director cu acelaşi nume ca cel al pachetului şi care să poate fi "accesat" din cadrul variabilei de sistem `CLASSPATH`.

Exerciţii

Pe scurt

1. Ce înseamnă ascunderea informaţiei? Ce este încapsularea? Cum suportă Java aceste concepte?

2. Explicaţi secţiunile `public` şi `private` ale unei clase.

3. Descrieţi rolul unui constructor.

4. Dacă o clasă nu are nici un constructor, ce va face compilatorul?

5. Ce înseamnă accesul de tip "package friendly"?

Teorie

1. Să presupunem că funcţia `main` din **Listing 4.2**, ar fi făcut parte din clasa `Circle`.

   - Ar fi funcţionat programul?
   - Am fi putut decomenta linia 15 din metoda `main()` fără a genera erori?

În practică

1. Creaţi o clasă cu un constructor implicit (fără parametri) care afişează un mesaj. În cadrul metodei `main()`, creaţi o instanţă a clasei şi verificaţi dacă mesajul a fost afişat.

Digitized by Google

2. Adăugaţi clasei precedente un constructor care primeşte un parametru de tip `String`, şi care afişează parametrul primit alături de mesajul anterior. Creaţi câte o instanţă a clasei folosind ambii constructori şi verificaţi mesajele afişate.

3. Creaţi un şir de obiecte de tipul celor de la exerciţiul anterior, fără a crea însă obiectele pe care le referă elementele şirului. Verificaţi dacă mesajele din constructor sunt afişate. Încercaţi acum să creaţi câte un obiect pentru fiecare element din şir şi observaţi diferenţele.

4. Definiţi o clasă fără nici un constructor iar apoi în `main()` creaţi o instanţă a clasei respective pentru a verifica faptul că se creează automat un constructor implicit. Adăugaţi apoi clasei un constructor cu un parametru de tip `int`, lasând metoda `main()` nemodificată. Programul nu se va mai compila. De ce?

5. Creaţi o clasă fără nici un constructor care să aibă un atribut de tip `int`, unul de tip `float`, unul de tip `boolean` şi unul de tip `String`. Creaţi o instanţă a clasei şi afişaţi valorile acestor atribute pentru a verifica iniţializările implicite.

6. Un lacăt cu cifru are următoarele proprietăţi: combinaţia cifrelor (o secvenţă de trei numere) este ascunsă; lacătul poate fi deschis prin furnizarea combinaţiei corecte; combinaţia poate fi modificată, dar numai de către cineva care cunoaşte combinaţia corectă. Proiectaţi o clasă care să definească metodele publice `open()` şi `changeCombo()` şi câmpuri de tipul `private` care să reţină combinaţia. Combinaţia va trebui stabilită în constructor.

7. Scrieţi o metodă care crează şi iniţializează o matrice de `int` cu valori aleatoare. Dimensiunile matricei, precum şi intervalul în care pot lua valori elementele matricei sunt date ca parametru. Adăugaţi o a doua metodă care afişează pe ecran matricea linie cu linie. Verificaţi cele două metode în `main()` iniţializând şi afişând matrice de diverse dimensiuni. Încercaţi apoi să extindeţi clasa pentru matrice cu trei dimensiuni.

8. Creaţi o clasă care să conţină membri de tip `private`, `public` şi friendly. Creaţi apoi în cadrul altei clase o instanţă a clasei şi observaţi mesajele de eroare pe care le obţineţi de la compilator încercând accesarea diverşilor membri. Ţineţi cont de faptul că clasele care nu sunt declarate ca făcând parte dintr-un anumit pachet, fac toate parte din acelaşi pachet implicit. Încercaţi acum să plasaţi cele două clase în pachete

Digitized by Google

diferite şi observaţi diferenţele între mesajele de eroare afişate la compilare.

Proiecte de programare

1. Scrieţi o clasă care suportă numere raţionale. Atributele clasei ar trebui să fie două variabile de tip `long`, una pentru numitor şi alta pentru numărător. Stocaţi numărul sub formă de fracţie ireductibilă, cu numitorul pozitiv. Furnizaţi un număr rezonabil de constructori, precum şi metodele `add()`, `subtract()`, `multiply()`, `divide()`; de asemenea, metodele `toString()`, `equals()` şi `compareTo()` (care se comportă ca cea din clasa `String`). Asiguraţi-vă că `toString()` funcţionează şi în cazul în care numitorul este zero.

2. Implementaţi o clasă pentru numere complexe, `Complex`. Adăugaţi aceleaşi metode ca şi la clasa `Rational`, dacă au sens (de exemplu, metoda `compareTo()` nu are sens aici). Adăugaţi accesori pentru partea reală şi cea imaginară.

3. Implementaţi o clasă completă `IntType` care să suporte un număr rezonabil de constructori şi metodele `add()`, `subtract()`, `multiply()`, `divide()`, `equals()`, `compareTo()` şi `toString()`. Menţineţi `IntType` sub forma unui şir de cifre suficient de mare. Pentru această clasă operaţia dificilă este împărţirea, urmată îndeaproape de înmulţire.

4. Implementaţi o clasă simplă numită `Date`. Clasa va trebui să reprezinte orice dată între 1 Ianuarie 1800 şi 31 Decembrie 2500; scădeţi două date; incrementaţi o dată cu un număr de zile; comparaţi două date folosind metodele `equals()` şi `compareTo()`. O dată este reprezentată intern ca numărul de zile care au trecut de la o anumită dată, care în cazul nostru este prima zi din 1800. Având în vedere acest lucru, toate metodele, mai puţin constructorul şi `toString()` vor fi banale.

   Indicaţie
   Regula pentru ani bisecţi este: un an este bisect dacă este divizibil prin 4, dar nu prin 100 sau dacă este divizibil cu 400. Astfel 1800, 1900, 2100 nu sunt ani bisecţi, dar 2000 este. Constructorul trebuie să verifice validitatea datei, ca şi `toString()`. Data ar putea deveni incorectă dacă operatorul de incrementare sau scădere ar face-o să iasă din limitele specificate.
   Odată ce am decis asupra specificaţiei, putem trece la implementare.

Digitized by Google

Partea oarecum dificilă este conversia între reprezentarea internă şi cea externă a datei. Urmează acum un posibil algoritm.
Creaţi două şiruri care să fie atribute statice. Primul şir, pe care îl putem numi `zilePanaLaIntaiALunii` va avea 12 elemente, reprezentând numărul de zile până la prima zi din fiecare lună a unui an nebisect. Astfel, el va conţine 0, 31, 59, 90 etc. Cel de-al doilea şir, `zilePanaLa1Ian` va conţine numărul de zile până la începutul fiecărui an, începand cu 1800. Astfel, el va conţine 0, 365, 730, 1095, 1460, 1826 etc, deoarece 1800 nu este an bisect, dar 1804 este. Programul va trebui să iniţializeze acest şir o singură dată folosind un iniţializator static. Veţi putea apoi folosi acest şir pentru conversia din reprezentarea internă în reprezentarea externă.

Digitized by Google

5. Moştenire

> Ştiinţă este orice disciplină în care nebunul generaţiei actuale poate să treacă dincolo de punctul atins de geniul generaţiei anterioare.
>
> Max Gluckman

Aşa cum am menţionat în capitolul anterior, unul dintre principalele scopuri ale programării orientate pe obiecte este reutilizarea codului. La fel cum inginerii folosesc aceleaşi componente din nou şi din nou în proiectarea circuitelor, programatorii au posibilitatea să refolosească şi să extindă obiectele, în loc să le reimplementeze. În limbajele de programare orientate pe obiecte, mecanismul fundamental pentru refolosirea codului este *moştenirea*. Moştenirea ne permite să extindem funcţionalitatea unui obiect. Cu alte cuvinte, putem crea noi obiecte cu proprietăţi extinse (sau restrânse) ale tipului original, formând astfel o ierarhie de clase. De asemenea, aşa cum vom vedea pe parcursul acestui capitol, moştenirea este mecanismul pe care Java îl foloseşte pentru a implementa metode şi clase generice (vezi subcapitolul 5.6).

În acest capitol vom prezenta:

- Principiile generale ale moştenirii, inclusiv polimorfismul;
- Cum este implementată moştenirea în Java;
- Cum poate fi derivată o colecţie de clase dintr-o singură clasă abstractă;
- *Interfaţa*, care este un caz particular de clasă;
- Cum se poate realiza programarea generică în Java folosind interfeţe;
- Ce sunt şi cum se folosesc clasele interioare;

Digitized by Google

- Cum se pot identifica tipurile de date în faza de execuţie a programului (engl. RunTime Type Identification, RTTI).

5.1 Ce este moştenirea?

Moştenirea este principiul fundamental al programării orientate pe obiecte care permite refolosirea codului între clasele înrudite. Moştenirea modelează relaţii de tipul ESTE-UN (sau ESTE-O). Într-o relaţie de tip ESTE-UN, spunem despre clasa derivată că ESTE-O variaţiune a clasei de bază. De exemplu, `Cerc` ESTE-O `Curba`, iar `Masina` ESTE-UN `Vehicul`. În schimb, o `Elipsa` NU-ESTE-UN `Cerc`. Relaţiile de moştenire formează *ierarhii*. De exemplu, putem extinde clasa `Masina` la `MasinaStraina` (pentru care se plăteşte vamă) şi `MasinaAutohtona` (pentru care nu se plăteşte vamă) etc.

Un alt tip de relaţie între obiecte este relaţia ARE-UN sau ESTE-COMPUS-DIN. De exemplu, `Masina` ARE-UN `Volan`. Această relaţie nu are proprietăţile standard dintr-o ierarhie de moştenire, motiv pentru care nu trebuie modelată prin moştenire, ci prin *agregare*, componentele devenind simple câmpuri de tip `private` ale clasei.

Chiar şi J2SDK foloseşte din plin moştenirea pentru a-şi implementa propriile biblioteci de clase. Un exemplu relativ familiar îl constituie *excepţiile*, prezentate pe larg în capitolul următor. Java defineşte clasa `Exception`. Aşa cum am văzut deja, există mai multe tipuri de excepţii, printre care putem aminti `NullPointerException` şi `ArrayIndexOutOfBoundsException`. Fiecare excepţie constituie o clasă separată, dar ele au totuşi trăsături comune, cum ar fi de exemplu metodele `toString()` sau `printStackTrace()`, utilizate pentru a furniza mesaje de eroare utile în depanare.

Moştenirea modelează în acest caz o relaţie de tip ESTE-UN. De exemplu, `NullPointerException` ESTE-O `Exception`. Datorită relaţiei de tip ESTE-UN, proprietatea fundamentală a moştenirii garantează că orice metodă aplicabilă lui `Exception` poate fi aplicată şi lui `NullPointerException`. Mai mult decât atât, un obiect de tip `NullPointerException` poate să fie referit de către o referinţă de tip `Exception` (reciproca nu este adevărată!). Prin urmare, deoarece metoda `toString()` este o metodă disponibilă în clasa `Exception`, vom putea întotdeauna scrie:

```
public void printException(Exception e)
{
  if (e instanceof NullPointerException)
  {
    System.out.println(e.toString());
  }
```


Digitized by Google

```
}
```

Dacă în faza de execuţie `e` referă un obiect `NullPointerException`, atunci `e.toString()` este un apel corect. Funcţie de modul de implementare al ierarhiei de clase, metoda `toString()` ar putea fi invariantă sau ar putea fi *specializată* pentru fiecare clasă distinctă. Atunci când o metodă este invariantă în cadrul unei ierarhii, adică are aceeaşi funcţionalitate pentru toate clasele din ierarhie, nu mai este necesar să rescriem implementarea acelei metode pentru fiecare clasă (ea este pur şi simplu moştenită).

Apelul lui `toString()` din exemplul de mai sus mai ilustrează un alt principiu important al programării orientate pe obiecte, cunoscut sub numele de *polimorfism*. O variabilă referinţă care este polimorfică poate să refere obiecte de tipuri diferite. Atunci când se aplică o metodă referinţei, se selectează automat operaţia adecvată pentru tipul obiectului care este referit în acel moment. În Java, toate tipurile referinţă sunt polimorfice. De exemplu, să presupunem că avem o referinţă `e` de tip `Exception`, căreia i se aplică `toString()`. În momentul execuţiei, maşina virtuală Java va vedea care este obiectul efectiv referit de `e` (care poate fi de exemplu `NullPointerException` sau `ArithmeticException`) şi va apela metoda `toString()` a acelui obiect. Acest proces este cunoscut sub numele de *legare târzie* sau *legare dinamică* (engl. "late binding"). Mai multe elemente despre polimorfism vom prezenta în paragraful 5.2.3.

În cazul moştenirii avem o clasă de bază din care sunt derivate alte clase. Clasa de bază reprezintă temelia principiului de moştenire. O *clasă derivată* moşteneşte toate proprietăţile clasei de bază, însemnând că toţi membrii publici ai clasei de bază devin membri publici ai clasei derivate (şi membrii `private` sunt moşteniţi, dar aceştia nu sunt accesibili în mod direct). Clasa derivată poate să adauge noi atribute şi metode şi poate modifica semnificaţia metodelor moştenite. Fiecare clasă derivată este o clasă complet nouă. Clasa de bază nu este în nici un fel afectată de modificările aduse în clasele derivate. Astfel, la crearea clasei derivate este imposibil să se strice ceva în clasa de bază.

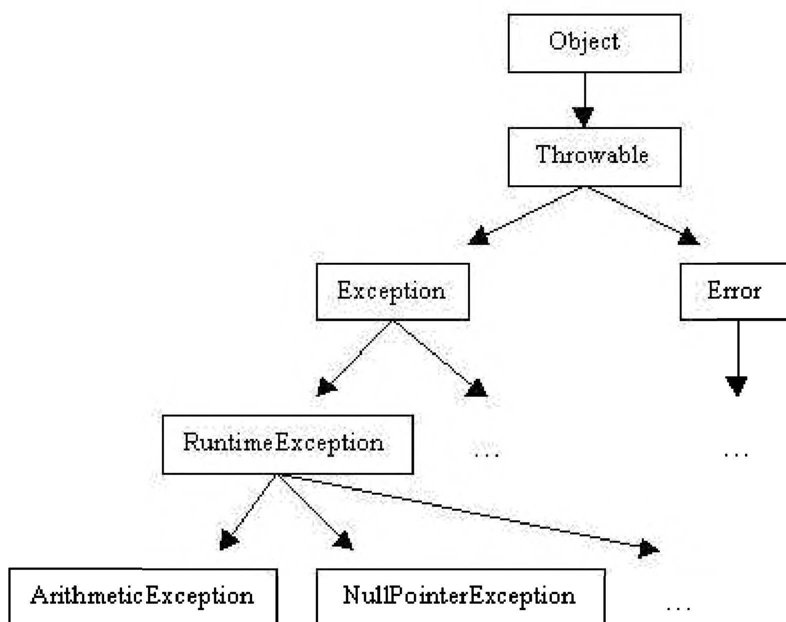
O clasă derivată este compatibilă ca tip cu clasa de bază, ceea ce înseamnă că o variabilă referinţă de tipul clasei de bază poate referi un obiect al clasei derivate, dar nu şi invers. Clasele surori (cu alte cuvinte clasele derivate dintr-o clasă comună) nu sunt compatibile ca tip.

Aşa cum am menţionat anterior, folosirea moştenirii generează de obicei o ierarhie de clase. **Figura 5.1** prezintă o mică parte din ierarhia de excepţii a limbajului Java. Remarcaţi faptul că `NullPointerException` este indirect derivată din `Exception`. Acest lucru nu constituie nici o problemă, deoarece relaţiile de tipul ESTE-UN sunt tranzitive. Cu alte cuvinte, dacă X ESTE-UN

Digitized by Google

Y şi Y ESTE-UN Z, atunci X ESTE-UN Z. Ierarhia `Exception` ilustrează în acelaşi timp designul clasic în care se extrag caracteristicile comune în clasa de bază, urmând ca specializările să fie adăugate în clasele derivate. Într-o astfel de ierarhie, clasa derivată este o *subclasă* a clasei de bază, iar clasa de bază este o *superclasă* a clasei derivate. Aceste relaţii sunt tranzitive; mai mult, operatorul `instanceof` funcţionează pentru subclase. Astfel, dacă `obj` este de tipul `X` (şi nu e `null`), atunci expresia `obj instanceof Z` este adevărată.

Figura 5.1: O parte a ierarhiei de excepţii a limbajului Java



În următoarele secţiuni vom examina următoarele probleme:

- Care este sintaxa folosită pentru a deriva o clasă nouă dintr-o clasă de bază?
- Cum afectează acest lucru statutul membrilor `private` sau `public`?
- Cum precizăm faptul că o metodă este invariantă pentru o ierarhie de clase?

Digitized by Google

- Cum factorizăm aspectele comune într-o clasă abstractă, pentru a crea apoi o ierarhie?
- Putem deriva o clasă nouă din mai multe clase (moştenire multiplă)?
- Cum se foloseşte moştenirea pentru a implementa codul generic?

Câteva dintre aceste subiecte sunt ilustrate prin implementarea unei clase `Shape` din care vom deriva clasele `Circle`, `Square` şi `Rectangle`. Ne vom folosi de acest exemplu pentru a vedea modalitatea în care Java implementează polimorfismul, dar şi pentru a vedea cum poate fi folosită moştenirea pentru a implementa metode generice.

5.2 Sintaxa de bază Java

O clasă derivată moşteneşte toate proprietăţile clasei de bază. Clasa derivată poate apoi să adauge noi atribute, să redefinească metode sau să adauge noi metode. Fiecare clasă derivată este o clasă complet nouă. Aspectul general al unei clase derivate este prezentat în **Listing 5.1**. Clauza `extends` declară faptul că o clasă este derivată dintr-o (extinde o) altă clasă.

Listing 5.1: Aspectul general al unei clase derivate

```java
public class Derived extends Base
{
    //membrii (public sau protected) ai clasei Base care
    //nu sunt listati mai jos vor fi mosteniti nemodificati,
    //cu exceptia constructorului;

    //membri public ai clasei Derived;
    //constructori, daca cel implicit nu este suficient;
    //metode din Base a caror implementare este
    //modificata;
    //metode publice aditionale;

    //membri private ai clasei Derived;
    //atribute si metode private;
}
```

Iată o scurtă descriere a unei clase derivate:

- În general, toate atributele sunt `private`, deci atributele suplimentare vor fi adăugate clasei derivate prin precizarea lor în secţiunea `private`;
- Orice metodă non-`private` din clasa de bază care nu este precizată în clasa derivată este moştenită nemodificat, cu excepţia constructorilor;

Digitized by Google

- Orice metodă din clasa de bază care este definită în secţiunea `public` a clasei derivate este redefinită. Noua metodă va fi aplicată obiectelor din clasa derivată;

- Metodele `public` din clasa de bază nu pot fi redefinite în secţiunea `private` a clasei derivate;

- Clasei derivate îi putem adăuga metode suplimentare.

5.2.1 Reguli de vizibilitate

Ştim deja că orice membru care este declarat `private` este accesibil doar în cadrul metodelor clasei. Rezultă deci că nici un membru `private` din clasa de bază nu va fi direct accesibil în clasa derivată (el va putea fi accesat prin intermediul modificatorilor şi accesorilor moşteniţi de la clasa de bază).

Există situaţii în care clasa derivată trebuie să aibă acces la membrii clasei de bază. Există două opţiuni pentru a realiza acest lucru. Prima este aceea de a utiliza accesul de tip `public` sau *friendly*. Totuşi, accesul de tip `public` permite accesul şi altor clase, pe lângă clasele derivate şi încalcă principiul ascunderii informaţiei. Pe de altă parte, accesul de tip *friendly* funcţionează doar dacă clasa derivată este în acelaşi pachet cu clasa de bază, lucru care nu este obligatoriu (vezi capitolul 4.4.4).

Dacă dorim să restrângem accesul unor membri, astfel încât ei să fie vizibili *doar* pentru clasele derivate, putem să îi declarăm ca fiind de tip `protected`. Un membru de tip `protected` este `private` pentru toate clasele cu excepţia claselor derivate şi a claselor din acelaşi pachet. În consecinţă vizibilitatea membrilor de tip `protected` este mai lejeră decât a membrilor de tip *package friendly*. Declararea atributelor ca fiind `public` sau `protected` încalcă principiile încapsulării şi ascunderii informaţiei, fiind folosită doar din motive de comoditate. De obicei, este preferabil să se scrie modificatori şi accesori (*setter*-i şi *getter*-i) sau să se folosească accesul de tip *friendly*. Totuşi, dacă o declaraţie `protected` vă scuteşte de scrierea de cod stufos, atunci se poate recurge la ea.

5.2.2 Constructor şi `super`

Fiecare clasă derivată trebuie să îşi definească proprii constructori. Dacă nu se scrie nici un constructor, Java va genera un constructor implicit fără parametri. Acest constructor va apela constructorul fără parametri al clasei de bază (dacă nu există, se obţine eroare la compilare!) pentru membrii care au fost

Digitized by Google

moşteniţi, după care va aplica iniţializarea implicită pentru atributele adăugate (adică 0/false pentru tipurile primitive şi `null` pentru tipurile referinţă).

Construirea unui obiect al unei clase derivate are loc prin construirea prealabilă a porţiunii moştenite (constructorul clasei derivate apelează automat constructorul implicit al clasei de bază). Acest lucru este natural, deoarece principiul încapsulării afirmă că partea moştenită este o entitate unică, iar constructorul clasei de bază ne spune cum să iniţializăm această entitate.

Constructorii clasei de bază pot fi apelaţi explicit prin metoda `super()`. Astfel, constructorul implicit pentru o clasă derivată are de fapt forma:

```
public Derived()
{
    super();
}
```

Metoda `super` poate fi utilizată şi pentru a apela constructori cu parametri. De exemplu, dacă avem o clasă de bază care are un constructor cu doi parametri de tip `int`, atunci constructorul clasei derivate va avea în general forma:

```
public Derived(int x, int y)
{
    super(x, y);
    // alte instructiuni
}
```

Metoda `super()` poate să apară doar în *prima* linie dintr-un constructor. Dacă nu se face un apel explicit al lui `super()`, compilatorul va realiza automat un apel al metodei `super()` fără parametri chiar la începutul constructorului. Este important de remarcat că apelul implicit este către constructorul *implicit* (fără parametri) al clasei de bază. Dacă acest constructor nu există (să nu uităm că el se generează automat doar dacă **nu** există alţi constructori), se obţine eroare la compilare. De exemplu, clasa `Base` de mai jos defineşte un constructor cu un parametru de tip `int`:

```
public class Base
{
  public Base(int x)
  {
    System.out.println("Base(int) called.");
  }
}
```

Clasa `Derived`, extinde clasa `Base`, redefineşte constructorul cu parametru de tip `int` şi defineşte o metodă `main()` care creează un obiect de tip `Derived`:

```
public class Derived extends Base
{
```


Digitized by Google

```
  public Derived( int x )
  {
    System.out.println("Derived(int) called.");
  }

  public static void main(String[] args)
  {
    new Derived(2);
  }
}
```

Aparent, nimic nu este în neregulă cu aceste clase. Totuşi, construirea obiectului `Derived`, presupune construirea în prealabil a obiectului `Base`, pe care acesta îl extinde. Cum constructorul din `Derived` nu conţine un apel explicit al constructorului din `Base`, compilatorul va adăuga automat un apel către constructorul implicit. Dar, surpiză! Acest constructor nu există şi nici nu a fost generat implicit, deci se va obţine o eroare la compilare, în care suntem anunţaţi cu părere de rău că “nu a fost găsit constructorul `Base()`”. Corect este ca prima linie din constructorul clasei `Derived` să fie un apel către constructorul cu parametru de tip `int` din clasa `Base`:

```
public Derived( int x )
{
  super(x);
  System.out.println("Derived(int) called.");
}
```

La execuţia clasei `Derived` se va afişa în consolă:

```
Base(int) called.
Derived(int) called.
```

5.2.3 Redefinirea metodelor. Polimorfism

Polimorfismul, al treilea concept fundamental al programării orientate pe obiecte, alături de încapsulare şi moştenire, crează adeseori un anumit grad de confuzie în rândul programatorilor începători. Vom urmări în acest paragraf să clarificăm această importantă noţiune, lucru absolut necesar pentru o înţelegere profundă a programării orientate pe obiecte.

Capacitatea de a asuma diferite forme poartă numele de "polimorfism". Apa oferă un bun exemplu de polimorfism în lumea reală: apare sub formă solidă (gheaţa), lichidă, sau gazoasă (vaporii de apă). În Java, polimorfismul înseamnă că o singură variabilă referinţă poate fi folosită pentru a desemna mai multe obiecte - instanţe ale unor clase similare[1] - în diferite momente ale execuţiei

[1] a se citi clase derivate direct sau indirect din aceeaşi clasă de bază.

Digitized by Google

unui program. Când o referinţă `x` este folosită pentru a invoca o metodă a unui obiect (de exemplu, `x.metodaMea()`), metoda precisă care va fi apelată depinde de obiectul pe care variabila referinţă îl indică în acel moment.

O variabilă referinţă poate referi un obiect a cărui clasă este similară cu tipul variabilei referinţă. De exemplu, să presupunem că avem clasele `Animal`, `Pisica`, `Soarece`, `Caine`, definite ca mai jos:

Listing 5.2: Exemplu de polimorfism

```
public class Animal
{
    public String caracteristica()
    {
        return "necunoscuta";
    }
}

public class Soarece extends Animal
{
    public String caracteristica()
    {
        return "are culoarea gri";
    }
}

public class Pisica extends Animal
{
    public String caracteristica()
    {
        return "miauna";
    }
}

public class Caine extends Animal
{
    public String caracteristica()
    {
        return "latra";
    }
}

public class PolimorfismEx
{
    public static void main(String args[])
    {
        Animal a = new Soarece();
        System.out.println("Caracteristica primului animal: " +
                           a.caracteristica());

```


Digitized by Google

```
        //alte instructiuni...

        a = new Caine();
        System.out.println("Caracteristica celui de-al doilea" +
                           " animal: " + a.caracteristica());

        //alte instructiuni...
    }
}
```

Ceea ce atrage atenţia asupra acestui program este faptul că toate cele patru metode `caracteristica()` au aceeaşi semnătură (nume şi listă de parametri). Acest lucru este permis deoarece nu există două metode cu aceeaşi semnătură în *aceeaşi* clasă. Cum trei dintre aceste clase (`Soarece`, `Caine`, `Pisica`) extind o a patra clasă (`Animal`), spunem că metodele `caracteristica()` din clasele derivate *redefinesc* (engl. override) metoda `caracteristica()` din clasa de bază.

Să analizăm acum instrucţiunea următoare:

```
Animal a = new Soarece();
```

În această situaţie, variabila referinţă `a` care este de tipul `Animal` desemnează un obiect de tipul `Soarece`, care ESTE-UN `Animal` prin moştenire (prin urmare, atribuirea este corectă). Mai departe, întâlnim în cadrul aceleiaşi secvenţe de cod:

```
...

a = new Caine();
```

Cu alte cuvinte, variabila `a` referă (la momente diferite de timp) obiecte de tipuri diferite, dar similare (prin faptul că sunt derivate din aceeaşi clasă: `Animal`). Pentru a vă convinge de acest lucru, rulaţi acest program. El va afişa următorele date:

```
Caracteristica primului animal: are culoarea gri
Caracteristica celui de-al doilea animal: latra
```

După cum se poate observa, apelurile metodei `caracteristica()`, prin intermediul referinţei `a`, au determinat apelarea metodelor corespunzătoare din clasele `Soarece` şi `Caine` (şi prin urmare, două afişări diferite), deoarece la momentul fiecărui apel, variabila `a` referea un obiect de tip diferit. Poate că unii dintre dumneavoastră se aşteptau ca singura metodă apelată să fie cea din clasa `Animal`, motiv pentru care rezultatul vi se pare anormal, dar el va fi perfect explicabil după ce vom detalia noţiunea de apel polimorfic al unei metode.

Digitized by Google

Ca o paranteză, trebuie remarcat faptul că inversarea rolurilor celor două clase nu este posibilă în cadrul acestui exemplu (normal de altfel, deoarece un animal nu este în mod obligatoriu un şoarece):

```
Soarece s = new Animal(); //INCORECT...
```

După cum am văzut anterior, în cadrul subcapitolului 5.1, moştenirea permite tratarea unui obiect ca fiind de tipul propriu *sau* de tipul de bază (din care este derivat tipul propriu). Această caracteristică este esenţială deoarece permite mai multor tipuri (derivate din acelaşi tip de date) să fie tratate unitar, ca şi cum ar fi un singur tip, motiv pentru care aceeaşi secvenţă de cod va fi funcţională pentru toate aceste tipuri. Pe de altă parte, apelul polimorfic al unei metode permite unui tip să exprime distincţia faţă de un alt tip similar, atâta timp cât amândouă sunt derivate din aceeaşi clasă de bază. Distincţia este exprimată prin diferenţa în comportamentul metodelor care pot fi apelate prin intermediul clasei de bază (în exemplul nostru, este vorba despre metoda `caracteristica()`).

Polimorfismul permite scrierea de cod doar în raport cu clasa de bază, ca şi cum "am fi uitat" de clasele derivate. Totuşi, cei mai mulţi programatori, în special cei cu experienţă în programarea procedurală, întâmpină anumite dificultăţi în înţelegerea modului de funcţionare a polimorfismului.

Dificultatea poate fi întâlnită şi în cadrul programului din exemplul anterior. Afişarea celor două mesaje este evident cea dorită, deşi parcurgând codul sursă al programului, am putea avea impresia că programul nu va funcţiona în acest mod. Dacă privim mai atent linia 38, observăm că metoda `caracteristica()` este apelată prin intermediul unei referinţe de tipul `Animal`. Aşa că ne punem în mod logic întrebarea: este posibil să ştie compilatorul că variabila referinţă de tipul `Animal` referă un obiect de tipul `Soarece`, din moment ce se apelează metoda acesteia din urmă? De unde ştie compilatorul că este referit un obiect de tip `Soarece`, şi nu un obiect de alt tip (de exemplu, `Pisica` sau `Caine`)? Răspunsul este că, de fapt, compilatorul nu ştie acest lucru. Pentru a înţelege mai bine această problemă, este necesară o examinare mai atentă a noţiunii de *legare*.

Conectarea unui apel de metodă de un anumit corp de metodă poartă numele de *legare* (engl. binding). Când legarea are loc înainte de rularea programului respectiv (cu alte cuvinte, în faza de compilare), spunem că este vorba despre o *legare statică* (engl. early binding). Termenul este specific programării orientate pe obiecte. În programarea procedurală (gen programarea în C sau Pascal), noţiunea de legare statică nici nu există, pentru că toate legăturile se fac în mod static.

În programul anterior, compilatorul nu ştie ce metodă `caracteristica()`

Digitized by Google

(din clasa `Animal`, `Soarece`, `Pisica` sau `Caine`?) să apeleze, având doar o referinţă de tipul `Animal`. Soluţia constă însă în *legarea târzie* (engl. late binding). Cunoscută şi sub numele de *legare dinamică* sau *legare în faza de execuţie* (engl. dynamic binding, run-time binding), legarea târzie permite determinarea în faza de execuţie a tipului obiectului referit de o variabilă referinţă şi apelarea metodei specifice acestui tip (în exemplul nostru, metoda `caracteristica()` din clasa `Soarece`). Mecanismul de apelare a metodelor determină în mod corect metoda care trebuie apelată şi o apelează. Legarea este un fenomen care are loc automat - nu este atribuţia programatorului să decidă dacă o metodă este apelată prin legare statică sau dinamică.

Am ajuns la momentul în care trebuie să conturăm diferenţa dintre *redefinirea* (suprascrierea) unei metode şi *supraîncărcarea* ei. Cu alte cuvinte, *overriding vs. overloading*. Finalul capitolul 2, secţiunea 2.6, a prezentat noţiunea de supraîncărcare a metodelor. Recapitulând pe scurt informaţiile precizate acolo, supraîncărcarea metodelor permite existenţa în interiorul aceleaşi clase a mai multor metode cu acelaşi nume, dar cu lista *diferită* de parametri (prin urmare, cu semnături diferite). Astfel, putem avea o metodă `int max(int a, int b)` şi o metodă `int max(int a, int b, int c)`, ambele în cadrul aceleiaşi clase. Evident că, analog, putem avea prima metodă în cadrul unei clase de bază şi cea de-a doua în cadrul unei clase derivate din clasa de bază. Pe de altă parte, noţiunea de suprascriere (redefinire) se bazează tocmai pe ideea că metodele trebuie să aibe *aceeaşi* semnătură şi să se afle în clase derivate din clasa de bază (vezi exemplul din această secţiune). Diferenţa dintre cele două concepte este în acest moment clar definită.

Cele două concepte pot provoca uneori greşeli de programare greu detectabile, în cazul în care nu sunt folosite corect. Iată un exemplu sugestiv: să presupunem că metoda `caracteristica()` din clasa `Soarece` ar fi avut de fapt următoarea definire:

```
public String caracteristica(int i)
{
    return "are blana gri";
}
```

Poate că acest antet ar fi fost o greşeală neintenţionată a programatorului, sau pur şi simplu, ar fi fost ceva intenţionat. Ce s-ar fi întâmplat în această situaţie la execuţia liniei 38 din exemplul nostru? Cert este că aceasta nu ar fi fost interpretată drept o eroare de către compilator. Programul ar fi rulat şi ar fi afişat rezultatele:

```
Caracteristica primului animal: necunoscuta
```


Digitized by Google

```
Caracteristica celui de-al doilea animal: latra
```

Poate că vi se pare ciudat rezultatul execuţiei versiunii modificate a programului. Să vedem ce s-a întâmplat de fapt. Deoarece în clasa `Soarece`, noua metodă `caracteristica()` are o semnătură diferită faţă de metoda cu acelaşi nume a clasei de bază, am realizat o *supraîncărcare* a acelei metode, şi nu o *redefinire* a ei. Astfel clasa `Soarece` are acum două metode `caracteristica()` diferite: una moştenită prin derivarea din clasa de bază şi cealaltă prin supraîncărcarea metodei din clasa de bază. În urma apelului din linia 38 se va executa însă metoda moştenită din clasa de bază, şi nu cea *supraîncărcată*, deoarece aceea corespunde ca semnătură. Astfel, mesajul afişat este `"necunoscuta"`.

Deşi se intenţionase o redefinire a acelei metode, datorită faptului că s-a greşit la definirea antetului ei, rezultatul nu a fost cel aşteptat. Totuşi, compilatorul a presupus că intenţia a fost de a supraîncărca metoda şi nu de a o redefini, motiv pentru care nu a afişat nici o eroare. Evident că acesta nu este un apel polimorfic de metodă. Odată ce a fost depistată eroarea, ea poate fi înlăturată simplu, însă este infinit mai greu în cazul unor aplicaţii mai complexe. La fel de adevărat este faptul că uneori *exact* acest comportament este dorit. Programatorii avansaţi vor şti cu siguranţă cum să "jongleze" cu cele două noţiuni. Cei începători vor trebui să fie foarte atenţi în astfel de situaţii, pentru a implementa exact comportamentul pe care îl doresc.

Datorită polimorfismului, exemplul nostru poate fi extins, adăugând noi funcţionalităţi prin intermediul unor tipuri noi de date (clase) care moştenesc comportamentul clasei de bază. Altfel spus, putem adăuga clase noi ce extind comportamentul clasei `Animal`, fără ca aceasta să prejudicieze codul existent.

Referitor la modul de utilizare a polimorfismului există o singură problemă ce trebuie tratată cu mare atenţie pentru că este generatoare de mari probleme, greu detectabile mai ales în cadrul unor programe de dimensiuni mari: este vorba despre utilizarea metodelor polimorfice în cadrul constructorilor. Deoarece obiectele nu sunt complet construite în această fază a execuţiei unui program, ceea ce presupune că identificarea tipului obiectelor respective este problematică, apelul polimorfic al unor metode poate crea un comportament imprevizibil şi, în cele mai multe situaţii, nedorit.

Polimorfismul presupune existenţa unei "feţe" (funcţionalitatea comună din clasa de bază) şi a mai multor "forme" care folosesc această faţă (diferitele funcţionalităţi ale metodelor legate dinamic). Reţineţi că polimorfismul apare doar atunci când este vorba despre legarea târzie a metodelor. Polimorfismul oferă o nouă viziune asupra modului de decuplare dintre ceea *ce* face o clasă şi *cum* îndeplineşte ea acest lucru. Polimorfismul realizează o decuplare la nivel

Digitized by Google

de *tipuri*. De asemenea, permite o bună organizare a codului, dar şi crearea de programe extensibile în adevăratul sens al cuvântului, care pot fi îmbunătăţite ca funcţionalitate nu numai în faza de creare a aplicaţiei, ci şi pe parcursul existenţei ei, când noi facilităţi sunt de dorit.

5.2.4 Redefinirea parţială a metodelor

Am văzut deja că metodele din clasa de bază pot fi redefinite în clasele derivate prin furnizarea unei metode din clasa derivată care să aibă aceeaşi semnătură. De asemenea, metoda din clasa derivată nu are dreptul să adauge excepţii la lista `throws` (detalii despre `throws` în capitolul următor).

Adeseori, metoda din clasa derivată nu redefineşte complet metoda din clasa de bază, ci extinde operaţiile pe care aceasta le realizează. Acest proces este numit *redefinire parţială*. Cu alte cuvinte vrem să facem ceea ce face şi metoda din clasa de bază, plus încă ceva. Apeluri ale metodei din clasa de bază pot fi realizate folosind din nou `super`. Iată un exemplu:

Listing 5.3: Exemplu de redefinire parţială

```
public class Student extends Human
{
    public doWork()
    {
        takeBreak();        //pauzele lungi si dese
        super.doWork();     //cheia marilor
        takeBreak();        //succese
    }
}
```

Astfel, metoda `doWork()` din clasa `Student` extinde metoda `doWork()` din clasa `Human`, adăugându-i două elemente importante: pauza şi pauza... Apelul polimorfic se aplică în mod identic şi în cazul redefinirii parţiale.

5.2.5 Metode şi clase `final`

Aşa cum am precizat anterior, clasa derivată poate să redefinească sau să accepte nemodificate metodele din clasa de bază. În multe cazuri este clar faptul că o anumită metodă trebuie să fie invariantă de-a lungul ierarhiei de clase, ceea ce înseamnă că nici o clasă derivată nu trebuie să o redefinească. În acest caz, putem declara metoda ca fiind de tip `final`, iar ea nu va putea fi redefinită.

Pe lângă faptul că declararea unei metode `final` este o practică bună de programare, ea poate genera şi cod mai rapid. Declararea unei metode `final` (atunci când este cazul) este utilă deoarece intenţiile noastre devin astfel clare

Digitized by Google

pentru cititorul programului şi al documentaţiei şi, pe de altă parte, putem preveni redefinirea accidentală pentru o metodă care nu trebuie să fie redefinită.

Pentru a vedea de ce folosirea lui `final` poate conduce la cod mai eficient, să presupunem că avem o clasă de bază numită `Base` care defineşte o metodă finală `f()`, iar `Derived` este o clasă care extinde `Base`. Să considerăm metoda:

```
public void xxx(Base obj)
{
    obj.f();
}
```

Deoarece `f()` este o metodă `final`, nu are nici o importanţă dacă în momentul execuţiei, `obj` referă un obiect de tip `Base` sau un obiect de tip `Derived`; definirea lui `f` este invariantă, deci ştim de la început ceea ce `f` va face. O consecinţă a acestui fapt este că decizia pentru codul care va fi executat se ia încă de la *compilare*, şi nu la *execuţie*. Este vorba, în mod evident, de o *legare statică*. Deoarece legarea se face la compilare şi nu la execuţie, programul ar trebui să ruleze mai repede. Dacă acest fapt este sau nu perceptibil efectiv (ţinând cont de viteza de prelucrare a procesoarelor din generaţia actuală) depinde de numărul de ori în care evităm deciziile din timpul execuţiei programului. În Java, orice legare de metode este polimorfică, prin intermediul legării târzii, cu excepţia situaţiei în care metoda este declarată `final`.

Un corolar al acestei observaţii îl constituie faptul că dacă `f` este o metodă `final` banală, cum ar fi un accesor pentru un atribut, compilatorul ar putea să înlocuiască apelul lui `f()`, direct cu corpul funcţiei. Există foarte multe optimizări pe care le face compilatorul java pentru a mări eficienţa codului, iar aceasta este doar una dintre ele. Astfel, apelul funcţiei va fi înlocuit cu o singură linie care accesează un atribut, economisindu-se astfel timp (apelul unei metode şi transmiterea de parametri sunt operaţii destul de costisitoare). Dacă `f()` nu ar fi fost declarată `final`, acest lucru ar fi fost imposibil, deoarece `obj` ar fi putut referi un obiect al unei clase derivate, pentru care definirea lui `f()` ar fi putut fi diferită.

Este interesant de remarcat faptul că, deoarece metodele statice nu au un obiect care să le controleze, apelul lor este rezolvat încă de la compilare folosind legarea statică.

Similare metodelor `final` sunt clasele `final`. O clasă `final` nu mai poate fi extinsă. În consecinţă, toate metodele unei astfel de clase sunt automat metode `final`. Ca un exemplu, clasa `Integer` din pachetul `java.lang` este o clasă `final`. De remarcat faptul că dacă o clasă are doar membri `final`, ea nu este neapărat o clasă `final`.

Digitized by Google

5.2.6 Metode şi clase abstracte

Până acum am văzut faptul că unele metode sunt invariante de-a lungul ierarhiei de clase (metodele `final`), iar alte metode îşi modifică semnificaţia de-a lungul ierarhiei. O a treia posibilitate este ca o metodă din clasa de bază să aibă sens *doar* pentru clasele derivate, şi vrem ca ea să fie obligatoriu definită în clasele derivate. Totuşi, implementarea metodei nu are nici un sens pentru clasa de bază. În această situaţie, putem declara metoda ca fiind abstractă.

O *metodă abstractă* este o metodă care declară funcţionalităţi care vor trebui neapărat implementate până la urmă în clasele derivate. Cu alte cuvinte o metodă abstractă spune ceea ce obiectele derivate trebuie să facă. Totuşi, ea nu furnizează nici un fel de implementare, ci fiecare clasă derivată trebuie să vină cu propria implementare.

O clasă care are cel puţin o metodă abstractă este o *clasă abstractă*. Java pretinde ca toate clasele abstracte să fie definite explicit ca fiind abstracte. Atunci când o clasă derivată nu redefineşte o metodă abstractă, metoda va fi moştenită abstractă şi în clasa derivată. În consecinţă dacă o clasă care nu intenţionăm să fie abstractă, nu redefineşte toate metodele abstracte, compilatorul va detecta inconsistenţa şi va genera un mesaj de eroare.

Un exemplu simplu de clasă abstractă este clasa `Shape` (rom. formă sau curbă), pe care o vom folosi ca exemplu în cadrul acestui capitol. Din `Shape` vom deriva forme specifice cum ar fi `Circle` sau `Rectangle`. Putem deriva apoi `Square` ca un caz particular de `Rectangle`. **Figura 5.2** prezintă ierarhia de clase care rezultă.

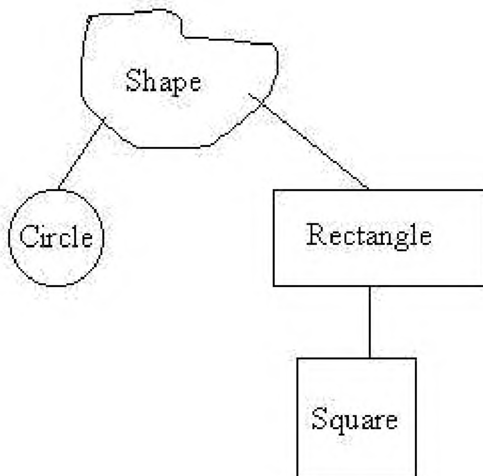
Clasa `Shape` poate să aibă membri care să fie comuni pentru toate clasele. Într-un exemplu mai extins, aceasta ar putea include coordonatele extremităţilor obiectului. Clasa ar putea declara şi defini metode cum ar fi `positionOf`, pentru a determina poziţia formei. Aceste metode sunt independente de tipul formei, motiv pentru care `positionOf` ar fi o metodă `final`. Clasa defineşte metode care se aplică fiecărui obiect în parte. Unele dintre aceste metode nu au nici un sens pentru clasa abstractă `Shape`. De exemplu, este dificil de calculat aria unui obiect oarecare; în consecinţă metoda `area` va fi declarată `abstract`.

Aşa cum am menţionat anterior, existenţa a cel puţin o metodă abstractă, face clasa să devină şi ea abstractă, deci ea nu va putea fi instanţiată. Astfel, nu vom putea crea un obiect de tip `Shape` şi vom putea crea doar obiecte derivate. Totuşi, ca de obicei, o referinţă de tip `Shape` poate să refere orice formă concretă derivată, cum ar fi `Circle` sau `Rectangle`. Exemplu:

```
Shape a,b;
a = new Circle(3.0);            //corect
```


Digitized by Google

Figura 5.2: Ierarhia de clase pentru forme




```
b = new Shape("circle");        //INCORECT!
```

Codul din **Listing 5.4** prezintă clasa abstractă `Shape`. La linia 13, declarăm o variabilă de tip `String` care reţine tipul formei, folosită doar în clasele derivate. Atributul este de tip `private`, deci clasele derivate nu au acces direct la el. Restul clasei cuprinde o listă de metode.

Constructorul nu va fi apelat niciodată direct, deoarece `Shape` este o clasă abstractă. Avem totuşi nevoie de un constructor care să fie apelat din clasele derivate pentru a iniţializa atributele `private`. Constructorul clasei `Shape` stabileşte valoarea atributului `name`.

Linia 15 declară metoda abstractă `area`. `area` este o metodă abstractă, deoarece nu putem furniza nici un calcul implicit al ariei pentru o clasă derivată care nu îşi defineşte propria metodă de calcul a ariei.

Metoda de comparaţie din liniile 22-25 nu este abstractă, deoarece ea poate fi aplicată în acelaşi mod pentru toate clasele derivate. De fapt, definirea ei este invariantă de-a lungul ierarhiei, de aceea am declarat-o `final`. Parametrul `rhs` (de la "right-hand-side") reprezintă un alt obiect de tip `Shape`, a cărui arie se compară cu cea a obiectului curent. Este interesant de remarcat faptul

Digitized by Google

că variabila `rhs` poate să refere orice instanţă a unei clase derivate din `Shape` (de exemplu o referinţă a clasei `Rectangle`). Astfel este posibil ca folosind această metodă să comparăm aria obiectului curent (care poate fi, de exemplu, o instanţă a clasei `Circle`) cu aria unui obiect de alt tip, derivat din `Shape`. Acesta este un exemplu excelent de folosire a polimorfismului.

Metoda `toString` din liniile 27-30 afişează numele formei şi aria ei. Ca şi metoda de comparaţie `lessThan`, ea este invariantă de-a lungul ierarhiei (nu face decât să concateneze două stringuri, indiferent de natura formei), de aceea a fost declarată *final*.

Listing 5.4: Clasa de bază abstractă `Shape`

```
1  /* Clasa de baza abstracta pentru forme
2   *
3   * CONSTRUIREA: nu este permisa, Shape fiind abstracta.
4   * Constructorul cu un parametru este furnizat ptr. clasele
5   * derivate.
6   *-------------------metode publice----------------------
7   * double area()         --> Intoarce aria (abstracta)
8   * boolean lessThan      --> Compara doua forme dupa arie
9   * String toString       --> Metoda uzuala pentru scriere
10  */
11 abstract class Shape
12 {
13   private String name;
14 
15   abstract public double area();
16 
17   public Shape(String shapeName)
18   {
19     name = shapeName;
20   }
21 
22   final public boolean lessThan(Shape rhs)
23   {
24     return area() < rhs.area();
25   }
26 
27   final public String toString()
28   {
29     return name + ", avand aria " + area();
30   }
31 }
```

Înainte de a trece mai departe, să rezumăm cele patru tipuri de metode ale unei clase:

1. *Metode finale*. Apelul lor este rezolvat încă de la compilare. Folosim

Digitized by Google

metode `final` doar atunci când metoda este invariantă de-a lungul ierarhiei (adică atunci când metoda nu este niciodată redefinită);

2. *Metode abstracte*. Apelul lor este rezolvat în timpul execuţiei. Clasa de bază nu furnizează nici o implementare a lor şi este abstractă. Clasele derivate trebuie fie să implementeze metoda, fie devin ele însele abstracte;

3. *Metode statice*. Apelul este rezolvat la compilare, deoarece nu există obiect care să le controleze;

4. *Alte metode*. Apelul este rezolvat la execuţie. Clasa de bază furnizează o implementare implicită care fie va fi redefinită (parţial sau total) în clasele derivate, fie acceptată nemodificată.

5.3 Exemplu: Extinderea clasei Shape

În această secţiune vom implementa clasele derivate din clasa `Shape` şi vom prezenta cum sunt ele utilizate într-o manieră polimorfică. Iată enunţul problemei:

Sortare de forme. *Se citesc N forme (cercuri, dreptunghiuri sau pătrate). Să se afişeze formele ordonate după arie.*

Implementarea claselor `Circle`, `Rectangle` şi `Square`, prezentată în **Listing 5.5** este simplă şi nu ilustrează aproape nici un concept pe care să nu-l fi prezentat deja. Singura noutate este că clasa `Square` este derivată din clasa `Rectangle` care este, la rândul ei, derivată din `Shape`. La implementarea fiecărei dintre aceste clase trebuie:

1. să definim un nou constructor;

2. să examinăm fiecare metodă care nu este `final` sau `abstract` pentru a vedea dacă dorim să o acceptăm nemodificată. Pentru fiecare astfel de metodă care nu corespunde cu necesităţile clasei trebuie să furnizăm o nouă definire;

3. să definim fiecare metodă abstractă;

4. să adăugăm alte metode dacă este necesar.

Digitized by Google

Listing 5.5: Codul complet pentru clasele `Circle`, `Rectangle` şi `Square`, care va fi salvat în 3 fişiere sursă separate

```
/* clasele Circle , Square si Rectangle ;
 * toate sunt derivate din Shape
 *
 * CONSTRUCTORI : ( a ) cu raza ptr . cerc , ( b ) cu latura
 * ptr . patrat , ( c ) cu lungime si latime ptr . dreptunghi
*-------------------metode publice-------------------
 * double area()-->Implementeaza metoda abstracta din Shape
 *
 * ATENTIE : Fisierul trebuie separat in 3 pentru compilare !
 */
public class Circle extends Shape
{
    private double radius ;

    public Circle ( double rad )
    {
      super ( " Circle " ) ;
      radius = rad ;
    }

    public double area ( )
    {
      return Math . PI * radius * radius ;
    }
 }

public class Rectangle extends Shape
{
    private double length ;
    private double width ;

    public Rectangle ( double len , double wid )
    {
      this ( len , wid , " Rectangle " ) ;
    }

    Rectangle ( double len , double wid , String name )
    {
     super ( name ) ;
     length = len ;
     width = wid ;
    }

    public double area ( )
    {
      return length * width ;
    }
```


Digitized by Google

```
}

public class Square extends Rectangle
{
  public Square(double side)
  {
    super(side, side, "Square");
  }
}
```

Pentru fiecare clasă am scris un constructor **public** simplu care permite inițializarea cu dimensiunile de bază (rază pentru cercuri, lungimea laturilor pentru dreptunghiuri și pătrate). Pentru clasa `Rectangle` a fost necesar și un constructor package-friendly, care să permită crearea unui obiect `Square`, cu numele corespunzător. Drept urmare, constructorul public al clasei `Rectangle` este și el adaptat, apelând de fapt constructorul package-friendly din aceeași clasă.

În constructorul public, vom inițializa mai întâi partea moștenită prin apelul lui `super`. În cazul clasei `Rectangle` acest apel nu este direct, ci prin intermediul constructorului package-friendly. Fiecare clasă trebuie să definească o metodă `area()`, deoarece `Shape` a declarat-o ca fiind abstractă. Dacă uităm să scriem o metodă `area` pentru una dintre clase, eroarea va fi detectată încă de la compilare, deoarece - dacă metoda `area()` lipsește - atunci clasa derivată este și ea abstractă. Observați că `Square` este dispusă să accepte metoda `area()` moștenită de la `Rectangle`, de aceea nu o mai redefinește. Logica acestei abordări derivă din faptul că aria unui pătrat se calculează exact ca și aria unui dreptunghi, prin produsul lungimii a două laturi alăturate.

După ce am implementat clasele, suntem gata să rezolvăm problema ordonării, iar pentru aceasta vom folosi un șir de clase `Shape`. Rețineți faptul că prin aceasta nu alocăm memorie pentru nici un obiect de tip `Shape` (ceea ce ar fi ilegal deoarece `Shape` este clasă abstractă), ci se alocă memorie doar pentru un șir de referințe către `Shape`. Aceste referințe vor putea referi obiecte de tip `Circle`, `Rectangle` sau `Square`[2].

În **Listing 5.6** realizăm exact acest lucru. Mai întâi citim obiectele. La linia 21, apelul lui `readShape()` constă în citirea unui caracter, urmată de dimensiunile figurii și de crearea unui nou obiect de tip `Shape`. **Listing 5.7** prezintă o implementare primitivă a acestei rutine. Observați că în cazul unei erori la citire se creează un cerc de rază 0 și se întoarce o referință la el. O soluție mai elegantă în această situație ar fi fost să definim și să aruncăm o excepție.

[2]Nici nu se poate inventa o ilustrare mai bună pentru polimorfism (poli = mai multe, morphos = forme). Într-adevăr referința de tip `Shape` (shape = formă) poate referi clase de mai multe (poli) forme (morphos): `Circle`, `Rectangle` și `Square`.

Digitized by Google

După aceasta, fiecare obiect creat de către `readShape()` este referit de către un element al şirului `shapes`. Se apelează `insertionSort()` (**Listing 5.8**) pentru a sorta formele. În final afişăm şirul rezultat de forme, apelând astfel implicit metoda `toString()`.

Listing 5.6: Rutina `main()` pentru citirea de figuri şi afişarea lor în ordine crescătoare

```
import java.io.* ;

class TestShape
{
  private static BufferedReader in;

  public static void main(String[] args)
  {
    try
    {
      //Citeste numarul de figuri
      in = new BufferedReader(new
      InputStreamReader(System.in));
      System.out.print("Numarul de figuri: ");
      int numShapes = Integer.parseInt(
      in.readLine());
      //citeste formele
      Shape[] shapes = new Shape[numShapes];
      for (int i = 0; i < numShapes; ++i)
      {
        shapes[i] = readShape();
      }

      //sortare si afisare
      insertionSort(shapes);
      System.out.println("Sortarea dupa arie: ");
      for (int i = 0; i < numShapes; ++i)
      {
        System.out.println(shapes[i]);
      }
    }
    catch(Exception e)
    {
      System.out.println(e);
    }
  }

  private static Shape readShape()
  {
    /*Implementarea in Listing 5.7*/
  }

```


Digitized by Google

```
  //sortare prin insertie
  private static void insertionSort(Shape[] a)
  {
    /* Implementarea in Listing 5.8 */
  }
}
```

Listing 5.7: Rutină simplă pentru citirea şi returnarea unei noi forme

```
//creaza un obiect adecvat de tip Shape functie de
//datele de intrare.
//utilizatorul introduce 'c', 's' sau 'r' pentru a indica
//forma, apoi introduce dimensiunile
//in caz de eroare se intoarce un cerc de raza 0

private static Shape readShape()
{
  double rad;
  double len;
  double wid;
  String s;

  try
  {
    System.out.println("Introduceti tipul formei: ");
    do
    {
      s = in.readLine();
    }while (s.length() == 0);

    switch (s.charAt(0))
    {
      case 'c':
        System.out.println("Raza cercului: ");
        rad = Integer.parseInt(in.readLine());
        return new Circle(rad);
      case 's':
        System.out.println("Latura patratului: ");
        len = Integer.parseInt(in.readLine());
        return new Square(len);
      case 'r':
        System.out.println("Lungimea si latimea "
        + " dreptunghiului pe linii separate: ");
        len = Integer.parseInt(in.readLine());
        wid = Integer.parseInt(in.readLine());
        return new Rectangle(len, wid);
      default:
        System.err.println("Tastati c, r sau s: ");
        return new Circle(0);
    }
```


Digitized by Google

```
    }
    catch(IOException e)
    {
        System.err.println(e);
        return new Circle(0);
    }
}
```

Listing 5.8: Sortarea prin inserţie

```
//sortare prin insertie
private static void insertionSort(Shape[] a)
{
    for (int p = 1; p < a.length; ++p)
    {
        Shape tmp = a[p];
        int j = p;
        for( ; j > 0 && tmp.lessThan(a[j-1]); --j)
        {
            a[j] = a[j-1];
        }
        a[j] = tmp;
    }
}
```

5.4 Moştenire multiplă

Toate exemplele prezentate până acum derivau o clasă dintr-o singură clasă de bază. În cazul *moştenirii multiple* o clasă este derivată din mai mult de o clasă de bază. De exemplu, putem avea clasele `Student` şi `Angajat`. Din aceste clase ar putea fi derivată o clasă `AngajatStudent`.

Deşi moştenirea multiplă pare destul de atrăgătoare, iar unele limbaje (cum ar fi C++) chiar o implementează, ea este îmbibată de subtilităţi care fac proiectarea claselor deosebit de dificilă. De exemplu, cele două clase de bază ar putea conţine metode care au aceeaşi semnătură, dar implementări diferite sau ar putea avea atribute cu acelaşi nume. Care dintre ele ar trebui folosit?

Din aceste motive, Java nu permite moştenirea multiplă. Java furnizează însă o alternativă pentru moştenirea multiplă prin intermediul conceptului de *interfaţă*.

Digitized by Google

5.5 Interfeţe

Interfaţa în Java este cea mai abstractă clasă posibilă. Ea constă doar din metode publice abstracte şi din atribute statice şi finale.

Spunem că o clasă *implementează* o anumită interfaţă dacă furnizează definiţii pentru toate metodele abstracte din cadrul interfeţei. O clasă care implementează o interfaţă se comportă ca şi când ar fi extins o clasă abstractă precizată de către acea interfaţă.

În principiu, diferenţa esenţială dintre o clasă abstractă şi o interfaţă este că, deşi amândouă furnizează o specificaţie a ceea ce clasele derivate trebuie să facă, interfaţa nu poate furniza nici un fel de detaliu de implementare sub formă de atribute sau de metode implementate. Consecinţa practică a acestui lucru este că derivarea din interfeţe nu suferă de problemele potenţiale pe care le are moştenirea multiplă, deoarece nu putem avea implementări diferite pentru aceeaşi metodă. Astfel, deşi o clasă poate să extindă o singură clasă, ea poate să implementeze mai mult de o singură interfaţă.

5.5.1 Definirea unei interfeţe

Din punct de vedere sintactic, nimic nu este mai simplu decât precizarea unei interfeţe. Interfaţa arată ca o declaraţie a unei clase, doar că foloseşte cuvântul cheie `interface`. Ea constă dintr-o listă de metode care trebuie declarate. Un exemplu de interfaţă este `Comparable`, prezentată în **Listing 5.9**.

Interfaţa `Comparable` precizează două metode pe care orice clasă derivată din ea trebuie să le implementeze: `compareTo()` şi `lessThan()`. Metoda `compareTo()` se va comporta similar cu metoda cu acelaşi nume din clasa `String`. Observaţi că nu este necesar să precizăm faptul că aceste metode sunt `public` sau `abstract`, deoarece acest lucru este implicit pentru metodele unei interfeţe.

Listing 5.9: Interfaţa `Comparable`

```
public interface Comparable
{
    int compareTo(Comparable rhs);
    boolean lessThan(Comparable rhs);
}
```

5.5.2 Implementarea unei interfeţe

O clasă implementează o interfaţă în doi paşi:

Digitized by Google

1. declară că implementează interfaţa;

2. defineşte implementări pentru toate metodele din interfaţă.

Un exemplu este prezentat în clasa din **Listing 5.10**, în care se defineşte clasa `MyInteger`. Clasa `MyInteger` are un comportament asemănător cu al clasei `Integer`, din pachetul `java.lang`.

În linia 1 se observă că atunci când implementăm o interfaţă folosim cuvântul cheie `implements` în loc de `extends`. În această clasă putem scrie orice metode dorim, dar trebuie să definim cel puţin metodele din interfaţă. Interfaţa este implementată în liniile 27-36. Remarcaţi faptul că trebuie să implementăm exact metodele precizate în cadrul interfeţei (`compareTo`, `lessThan`). Din acest motiv aceste metode au ca parametru un obiect de tip `Comparable` şi nu un `MyInteger`.

O clasă care implementează o interfaţă poate fi extinsă cu condiţia să nu fie finală. Astfel, dacă nu am fi declarat clasa `MyInteger` ca fiind finală, am fi putut-o extinde. O clasă care implementează o interfaţă poate totuşi să extindă şi o altă clasă. De exemplu, am fi putut, în principiu, scrie:

```
public class MyInteger extends Integer implements Comparable
```

Acest cod este incorect doar pentru că `Integer` este o clasă finală care nu poate fi astfel extinsă.

Listing 5.10: Clasa `MyInteger` care implementează interfaţa `Comparable`

```
1 final public class MyInteger implements Comparable
2 {
3   //constructor
4   public MyInteger(int value)
5   {
6     this.value = value;
7   }
8
9   //cateva metode
10  public String toString()
11  {
12    return Integer.toString(value);
13  }
14
15  public int intValue()
16  {
17    return value;
18  }
19
20  public boolean equals(Object rhs)
```


Digitized by Google

```
    {
        return rhs instanceof MyInteger &&
        value == ((MyInteger) rhs).value;
    }

    //implementarea interfetei
    public boolean lessThan(Comparable rhs)
    {
        return value < ((MyInteger) rhs).value;
    }

    public int compareTo(Comparable rhs)
    {
        return value < ((MyInteger) rhs).value ? -1 :
        value == ((MyInteger) rhs).value ? 0 : 1;
    }

    private int value;
}
```

5.5.3 Interfeţe multiple

Aşa cum am menţionat mai devreme, o clasă poate să implementeze mai mult de o singură interfaţă. Sintaxa pentru a realiza acest lucru este simplă. O clasă poate implementa mai multe interfeţe prin:

1. precizarea interfeţelor pe care le implementează;
2. implementarea tuturor metodelor din interfeţe.

Interfaţa este cea mai abstractă clasă posibilă şi reprezintă o soluţie elegantă la problema moştenirii multiple.

5.6 Implementarea de componente generice

Să ne reamintim că unul dintre scopurile principale ale programării orientate pe obiecte este suportul pentru reutilizarea codului. Unul dintre mecanismele importante folosite pentru îndeplinirea acestui scop este *programarea generică*: dacă implementarea unei metode este identică pentru mai multe clase (cu excepţia tipului de bază al obiectului), se poate folosi o *implementare generică* pentru a descrie funcţionalitatea de bază. De exemplu, putem scrie o metodă care să sorteze un şir de elemente. *Algoritmul* pentru această metodă este independent de tipul de obiecte care sunt sortate, deci putem folosi un algoritm generic.

Digitized by Google

Spre deosebire de multe dintre limbajele de programare mai noi (cum ar fi C++) care utilizează *şabloane* (template) pentru a implementa programarea generică, Java nu oferă suport pentru implementarea directă a programării generice, deoarece programarea generică poate fi implementată folosind doar conceptele de bază ale moştenirii. În această secţiune vom prezenta cum pot fi implementate metode şi clase generice în Java folosind principiile de bază ale moştenirii.

Ideea de bază în Java este că putem implementa o clasă generică folosind o superclasă adecvată, cum ar fi `Object`. În Java, dacă o clasă nu extinde o altă clasă, atunci ea extinde implicit clasa `Object` (din pachetul `java.lang`). Ca o consecinţă, fiecare clasă este o subclasă a lui `Object`.

Să considerăm clasa `MemoryCell` din **Listing 5.11**. Această clasă poate să reţină un obiect de tip `Object`. Deoarece `Object` este clasă de bază pentru orice clasă din Java, rezultă că această clasă poate să stocheze orice fel de obiecte.

Listing 5.11: Clasa generică `MemoryCell`

```
public class MemoryCell
{
  private Object storedValue;

  public Object read()
  {
    return storedValue;
  }

  public void write(Object x)
  {
    storedValue = x;
  }
}
```

Există două detalii care trebuie luate în considerare atunci când folosim această strategie. Ambele sunt ilustrate în **Listing 5.12**. Funcţia `main()` scrie valoarea 5 într-un obiect `MemoryCell`, după care citeşte din obiectul `MemoryCell`. În primul rând, tipurile primitive nu sunt obiecte. Astfel, `m.write(5)` ar fi fost incorect. Totuşi, aceasta nu este o problemă, deoarece Java dispune de clase *wrapper* (de "împachetare") pentru cele opt tipuri primitive. În loc să stocăm direct numărul 5, am stocat un obiect de tip `Integer` cu valoarea 5.

Al doilea detaliu este că rezultatul lui `m.read()` este un `Object`. Deoarece în clasa `Object` este definită o metodă `toString()`, nu este necesar să facem conversia de la `Object` la `Integer`. Referinţa returnată de

Digitized by Google

`m.read()` (de tip `Object`) este polimorfică şi ea referă de fapt un obiect de tip `Integer`. În consecinţă, se va apela automat metoda `toString()` a clasei `Integer`.

Dacă am fi vrut totuşi să extragem valoarea reţinută în obiectul de tip `MemoryCell`, ar fi trebuit să scriem o linie de genul

```
int i = ((Integer) m.read()).intValue();
```

în care se converteşte mai întâi valoarea returnată de `read` la `Integer`, după care se foloseşte metoda `intValue()` pentru a obţine un `int`.

Deoarece clasele *wrapper* sunt clase finale, constructorul şi accesorul `intValue()` pot fi expandate *inline*[3] de către compilator, generând astfel un cod la fel de eficient ca utilizarea directă a unui `int`.

Listing 5.12: Folosirea clasei generice `MemoryCell`

```
public class TestMemoryCell
{
  public static void main(String[] args)
  {
    MemoryCell m = new MemoryCell();

    m.write(new MyInteger(5));
    System.out.println("Continutul este: " + m.read());
  }
}
```

Un al doilea exemplu este problema sortării. Anterior am prezentat o metodă `insertionSort()` (**Listing 5.8**) care lucrează cu un şir de clase `Shape`. Ar fi interesant să rescriem această metodă pentru a putea sorta un şir generic. **Listing 5.13** prezintă o metodă de sortare generică `insertionSort()` care este practic identică cu metoda de sortare din **Listing 5.8** doar că foloseşte `Comparable` în loc de `Shape`. Putem pune această metodă într-o clasă `Sort`. Observaţi că nu sortăm `Object`, ci `Comparable`. Metoda `insertionSort()` sortează un şir de elemente `Comparable`, deoarece foloseşte `lessThan()`. Aceasta înseamnă că doar clasele care implementează interfaţa `Comparable` pot fi sortate astfel. De remarcat faptul că `insertionSort()` nu poate să sorteze un şir de obiecte `Shape`, deoarece clasa `Shape` din **Listing 5.4** nu implementează interfaţa `Comparable`. Unul dintre exerciţii propune modificarea clasei `Shape` în acest sens.

Pentru a vedea cum poate fi folosită metoda generică de sortare vom scrie un program, prezentat în **Listing 5.14**, care citeşte un număr nelimitat de va-

[3] Deşi anumiţi termeni (de exemplu, wrapper, inline) sunt în limba engleză, am optat pentru păstrarea lor în formă originală, deoarece aşa s-au consacrat în literatura de specialitate. Considerăm că traducerea în limba română ar fi inoportună, deteriorând semnificaţia lor.

Digitized by Google

lori întregi, le sortează şi afişează rezultatul. Metoda `readIntArray()` a clasei `Reader` (prezentată în capitolul precedent) este folosită pentru a citi un şir de numere întregi. Transformăm apoi şirul citit într-un şir de elemente `MyInteger`, care implementează interfaţa `Comparable`. În linia 15 creăm şirul, iar în liniile 16-19 obiectele care sunt stocate în şir. Sortarea este realizată în linia 22. În final, afişăm rezultatele în liniile 26-29. De reţinut că metoda `toString()` este implicit apelată pentru clasa `MyInteger`.

Listing 5.13: Algoritm de sortare generic

```
//sortare prin insertie
public static void insertionSort(Comparable[] a)
{
    for(int p = 1; p < a.length; ++p)
    {
        Comparable tmp = a[p];
        int j = p;
        for( ; j > 0 && tmp.lessThan(a[j-1]); --j)
        {
            a[j] = a[j-1];
        }
        a[j] = tmp;
    }
}
```

Listing 5.14: Citirea unui şir de numere întregi şi ordonarea lui folosind un algoritm generic

```
1  import io.*; //pachet definit anterior
2  public class SortIns
3  {
4    //program de test care citeste numere intregi
5    //(cate unul pe linie), le ordoneaza si apoi le afiseaza
6
7    public static void main(String[] args)
8    {
9
10     //citeste un sir de intregi
11     System.out.print("Elementele sirului: ");
12     int [] sir = Reader.readIntArray();
13
14     //conversie la un sir de MyInteger
15     MyInteger[] sirNou = new MyInteger[sir.length];
16     for (int i = 0; i < sir.length; ++i)
17     {
18       sirNou[i] = new MyInteger(sir[i]);
19     }
20
21     //aplica metoda de sortare
```


Digitized by Google

```
    Sort.insertionSort(sirNou);

    //afiseaza rezultatul sortarii
    System.out.println("Rezultatul sortarii: ");
    for(int i = 0; i < sirNou.length; ++i)
    {
      System.out.println(sirNou[i]);
    }
  }
}
```

5.7 Clase interioare (inner classes)

Clasele interioare reprezintă un tip special de clase care, după cum le spune şi numele, sunt definite în interiorul altor clase. Clasele care includ clase interioare se numesc (evident) clase exterioare. Clasele interioare sunt o facilitate importantă a limbajului Java, deoarece ne permit să grupăm clasele care aparţin în mod logic una de celalaltă şi să controlăm nivelul de vizibilitate între aceste clase. Clasele interioare reprezintă un concept distinct de compoziţia claselor (o clasă are un atribut de tipul altei clase). Diferenţele le veţi înţelege aprofundând treptat noţiunea de clasă interioară.

Necesitatea claselor interioare nu este deloc evidentă (cinstit vorbind, câţi dintre programatorii de Java care citesc această lucrare au auzit sau au folosit clase interioare?). Sperăm ca exemplul din această carte să clarifice utilitatea lor.

Aşa cum era de aşteptat, o clasă interioară se defineşte în cadrul altei clase care o cuprinde:

```
/**
 * Clasa exterioara.
 */
public class Scrisoare
{
    class Destinatie
    {
        ...
    }

    //alte posibile clase interioare
    ...

    //atribute si metode ale clasei exterioare
    ...
}
```


Digitized by Google

În exemplul de mai sus clasa exterioară este `Scrisoare`, iar cea interioară este `Destinatie`. Pentru a face acest exemplu funcţional, creaţi fişierul sursă `Scrisoare.java` ca în **Listing 5.15**.

Listing 5.15: Clasa `Scrisoare` care cuprinde clasa interioară `Destinatie`

```
/**
* Clasa exterioara Scrisoare
*/
public class Scrisoare
{
  /**
  * Clasa interioara Destinatie.
  */
  class Destinatie
  {
    /** Destinatia unei scrisori.*/
    private String dest;


    /** Creaza o destinatie pentru scrisoare.*/
    Destinatie(String dest)
    {
      this.dest = dest;
    }

    /** Obtine destinatia scrisorii.*/
    public String obtineDestinatia()
    {
      return dest;
    }
  }

  /** Trimite scrisoare la adresa specificata.*/
  public void trimiteScrisoare(String dest)
  {
    Destinatie d = new Destinatie(dest);

    System.out.println("Scrisoarea a fost trimisa " +
    "la destinatia: " + d.obtineDestinatia());
  }

  /** Programul principal.*/
  public static void main(String[] args)
  {
    //creaza o scrisoare
    Scrisoare s = new Scrisoare();

    //trimite scrisoarea in "SUA"
    s.trimiteScrisoare("SUA");
```


Digitized by Google

```
    }
}
```

Prin rularea acestui exemplu se va obţine următorul rezultat:

```
Scrisoarea a fost trimisa la destinatia: SUA
```

Dacă priviţi cu atenţie rezultatul compilării fişierului `Scrisoare.java`, veţi observa apariţia a două fişiere: `Scrisoare.class` şi `Scrisoare$Destinatie.class`. Cum rezultatul compilării fiecărei clase este un fişier cu extensia `.class`, care conţine bytecode-ul pentru clasa respectivă, este normal ca şi o clasă interioară să producă un fişier `.class`, care să conţină bytecode-ul produs de compilator. Pentru a face sesizabilă legătura dintre clasa exterioară şi cea interioară chiar şi în situaţia în care fişierele sursă ale claselor nu sunt disponibile, creatorii limbajului Java au ales ca fiecare clasă interioară să respecte o formulă strictă în ceea ce priveşte numele fişierului `.class` care îi corespunde. Această formulă arată astfel:

```
NumeClasaExterioara$NumeClasaInterioara
```

În exemplul nostru, clasa interioară `Destinatie`, care este utilizată în metoda `trimiteScrisoare()`, arată ca orice altă clasă. Singura diferenţă pare a fi faptul că este definită în interiorul altei clase. Vom vedea mai târziu că există şi alte diferenţe mai subtile.

În exemplul anterior clasa interioară `Destinatie` nu a fost utilizată în mod direct în programul principal, dar nimic nu ne împiedică să o utilizăm. Utilizarea în metoda `main` a clasei interioare se realizează astfel:

```
public static void main(String[] args)
{
        //creaza o scrisoare
        Scrisoare s = new Scrisoare();

        //trimite scrisoarea in "SUA"
        s.trimiteScrisoare("SUA");

        //creaza o alta destinatie
        Destinatie d = s.new Destinatie("Canada");
}
```

Sintaxa pentru crearea unei clase interioare este neobişnuită şi poate pune în încurcătură chiar şi programatorii care au ceva experienţă în Java. După cum se poate observa, pentru a crea direct o instanţă a clasei interioare (în exemplul nostru, `d`), este necesară utilizarea unei instanţe a clasei exterioare (în exemplul nostru, `s`). Astfel, nu este posibilă crearea unei instanţe a clasei interioare,

Digitized by Google

dacă nu este deja creată o instanţă a clasei exterioare. Limbajul Java impune o astfel de strategie, deoarece obiectul interior (cu alte cuvinte, instanţa clasei interioare) este conectat la obiectul exterior (instanţa clasei exterioare) din care este creat. Această conexiune este specifică doar claselor interioare şi va fi detaliată în cadrul acestui capitol.

Construcţia inestetică de instanţiere a clasei interioare poate fi totuşi evitată utilizând un mic artificiu. Vom adăuga în clasa exterioară `Scrisoare` o metodă `catre` care creează un obiect de tip destinaţie:

```
....

/** Stabileste destinatia unei scrisori.*/
public Destinatie catre(String dest)
{
    return new Destinatie(dest);
}

...

public static void main(String[] args)
{
    ...

    Destinatie d = s.catre("Canada");
}
```

O clasă interioară poate fi accesată uneori şi de alte clase, nu numai de către clasa exterioară care o conţine. Pentru a defini o astfel de referinţă a clasei interioare `Destinatie`, este necesar ca tipul referinţă asociat să conţină şi numele clasei exterioare, în formatul `ClasaExterioara.ClasaInterioara`. Pentru exemplul nostru declararea ar arăta astfel:

```
Scrisoare.Destinatie d;
```

Dacă necesitatea de a utiliza clasa interioară în cadrul altei clase apare prea frecvent, este un semn că am proiectat greşit aplicaţia, şi că, probabil, clasa respectivă nu este cazul să fie interioară.

Să revenim pentru moment asupra conexiunii care se realizează între clasa interioară şi cea exterioară. În momentul în care se crează o instanţă a clasei interioare, se crează o legătură între clasa interioară respectivă şi instanţa clasei exterioare, care conţine această clasă interioară. Acest lucru permite ca membrii clasei exterioare să fie accesibili clasei interioare, fără a folosi vreo calificare specială. De exemplu, putem modifica exemplul anterior, introducând un nou atribut pentru clasa exterioară, care va fi accesat în clasa interioară:

```
public class Scrisoare
```


Digitized by Google

```
{
    ...

    /**
     * Pentru a verifica daca scrisoarea a fost trimisa
     * la destinatie.
     */
    boolean trimisa;

    ...

    class Destinatie
    {
        ...

        Destinatie(String dest)
        {
            this.dest = dest;

            //initial scrisoarea nu a fost trimisa
            trimisa = false;
        }

        ...
    }
}
```

După cum se poate observa, atributul `trimisa` al clasei exterioare a fost utilizat în clasa interioară, fără nici o calificare specială, ca şi cum ar fi fost un atribut al acesteia. Explicaţia este următoarea: clasa interioară păstrează referinţa clasei exterioare, responsabilă de crearea instanţei clasei interioare (în exemplul nostru, această referinţă este `s`). Când un membru al clasei exterioare este utilizat în cadrul clasei interioare, referinţa aceasta (ascunsă) este folosită pentru a accesa acel membru. Din fericire pentru noi, compilatorul este cel care ţine cont de toate aceste detalii. Ca urmare, compilatorul va considera drept eroare situaţia în care nu va putea accesa referinţa clasei exterioare.

O altă caracteristică specifică doar claselor interioare este că ele pot fi ascunse complet de orice clasă, indiferent de pachetul în care se află. Spre deosebire de acest caz, mecanismul standard de ascundere a claselor permite doar ca o clasă să fie definită "prietenoasă" (friendly), ceea ce presupune că ea este totuşi accesibilă claselor din acelaşi pachet. Aşadar, dacă doriţi ca implementarea unei clase să fie complet ascunsă celorlalte clase, indiferent de pachetul în care se află, puteţi implementa clasa ca fiind interioară. Aşa cum era de aşteptat ascunderea unei clase interioare se face prin declararea ei ca fiind `private`:

```
private class Destinatie
{
```


Digitized by Google

```
    ...
}
```

În acest moment, clasa `Destinatie` este inutilizabilă pentru orice altă clasă cu excepţia clasei `Scrisoare` în care a fost definită. Pentru a verifica acest lucru, să mai adăugăm o clasă în fişierul `Scrisoare.java`:

```
public class Scrisoare
{
    ...
}

/** Clasa de test pentru clasa interioara privata.*/
class Test()
{
    public static void main(String[] args)
    {
        Scrisoare.Destinatie d; //EROARE DE COMPILARE
    }
}
```

Încercarea de a compila acest program este sortită eşecului, deoarece clasa interioară `Destinatie` reprezintă un membru privat al clasei `Scrisoare` şi nu poate fi accesat în afara clasei al cărei atribut este. Compilatorul va afişa o eroare de genul:

```
Scrisoare.java:59: Scrisoare.Destinatie has private
access in Scrisoare
                Scrisoare.Destinatie d;
                        ^
1 error
```

Deoarece clasa interioară este declarată ca fiind `private`, nici o clasă, cu excepţia clasei exterioare `Scrisoare`, nu o poate accesa. Cu alte cuvinte utilizarea clasei interioare este restricţionată total în afara clasei exterioare. Astfel, clasele interioare oferă o modalitate de a preveni orice dependenţă de alte clase, dar şi de a ascunde complet detaliile de implementare.

Analog o clasă interioară poate fi declarată ca fiind `protected`, ceea ce înseamnă că doar clasa exterioară şi clasele care o derivează pot accesa clasa interioară respectivă.

Un aspect important care trebuie reţinut este următorul: clasele normale (care nu sunt interioare) *nu* pot fi declarate `private` sau `protected`, ci doar `public` sau *friendly* (fără modificator de acces). De altfel, nici nu ar avea sens o clasă ne-interioară declarată în acel mod, deoarece nu am putea defini faţă de cine este `private` sau `protected`.

Digitized by Google

5.7.1 Clasificarea claselor interioare

În general, clasele interioare pe care le veţi folosi în programele dumneavoastră vor fi clase simple, uşor de înţeles şi utilizat, asemănătoare cu cele create în exemplele anterioare. Totuşi, limbajul Java oferă şi alte tipuri de clase interioare, ceva mai complexe.

În total, limbajul Java oferă patru tipuri de clase interioare:

- clase membre statice ale clasei exterioare

  Ca orice metodă statică a unei clase, o clasă membru static are acces la toate metodele şi atributele statice ale clasei părinte (cu alte cuvinte, ale clasei exterioare);

- clase membre nestatice ale clasei exterioare

  Spre deosebire de clasele membre statice, cele nestatice au acces la toate metodele şi atributele clasei exterioare, inclusiv la referinţa `this` a acesteia. Clasele interioare utilizate în exemplele precendente sunt de acest tip;

- clase locale

  Clasele locale sunt definite în cadrul unui bloc de cod, fiind vizibile doar în cadrul acelui bloc, similar cu o variabilă locală definită în cadrul unei metode;

- clase anonime

  Clasele anonime sunt clase locale fără nume.

5.7.2 Clasele membre statice ale clasei exterioare

Conexiunea dintre referinţa clasei exterioare şi cea a clasei interioare poate fi eliminată în situaţia în care clasa interioară este declarată statică. În cazul unei clase interioare statice:

- nu este necesară o instanţă a clasei exterioare pentru a crea o instanţă a clasei interioare;

- pot exista membri statici în clasa interioară respectivă.

Putem modifica exemplul nostru, pentru a utiliza clase interioare statice. Iată cum arată fişierul sursă `Scrisoare.java` în această situaţie:

Digitized by Google

Listing 5.16: Exemplu utilizare clase interioare statice

```
/**
 * Clasa exterioara Scrisoare
 */
public class Scrisoare
{
    /**
     * Clasa interioara Destinatie.
     */
    private static class Destinatie
    {
        /** Destinatia unei scrisori.*/
        private String dest;


        /** Creaza o destinatie pentru scrisoare.*/
        Destinatie(String dest)
        {
            this.dest = dest;
        }

        /** Obtine destinatia scrisorii.*/
        public String obtineDestinatia()
        {
            return dest;
        }

        /** Exemplu de membru static al clasei statice.*/
        public static void f()
        {
        }
    }

    public static Destinatie catre(String dest)
    {
        return new Destinatie(dest);
    }

    /** Trimite scrisoare la adresa specificata.*/
    public static void trimiteScrisoare(Destinatie d)
    {
        System.out.println("Scrisoarea a fost trimisa" +
              " la destinatia: " + d.obtineDestinatia());
    }

    /** Programul principal.*/
    public static void main(String[] args)
    {
        Destinatie d = catre("Canada");

```


Digitized by Google

```
        //trimite scrisoarea la destinatie
        trimiteScrisoare(d);

    }
}
```

Folosind clase statice interioare, nu a mai fost necesară crearea unei instanţe a clasei exterioare `Scrisoare`. După rularea programului se va afişa următorul rezultat:

```
Scrisoarea a fost trimisa la destinatia: Canada
```

5.7.3 Clasele membre nestatice ale clasei exterioare

Clasele membre nestatice au fost prezentate pe larg la începutul acestui subcapitol, motiv pentru care au mai rămas puţine detalii de adăugat:

- spre deosebire de clasele interioare statice, cele nestatice nu pot avea membri statici;
- uneori este necesară referinţa clasei exterioare, care este obţinută folosind sintaxa `NumeClasaExterioara.this`. În exemplul nostru, referinţa la clasa exterioară se obţine prin `Scrisoare.this`.

5.7.4 Clase locale

O clasă locală poate fi creată în cadrul unei metode, ca în exemplul următor:

Listing 5.17: Exemplu utilizare clase locale

```
/**
 * Clasa exterioara Scrisoare
 */
public class Scrisoare
{
    /** Metoda ce cuprinde o clasa interioara */
    public String catre(String dest)
    {
        /**
         * Clasa interioara Destinatie.
         */
        class Destinatie
        {
            /** Destinatia unei scrisori.*/
            private String dest;


```


Digitized by Google

```
            /** Creaza o destinatie pentru scrisoare.*/
            Destinatie(String dest)
            {
                this.dest = dest;
            }

            /** Obtine destinatia scrisorii.*/
            public String obtineDestinatia()
            {
                return dest;
            }
        }

        return (new Destinatie(dest)).obtineDestinatia();
    }

    /** Trimite scrisoare la adresa specificata.*/
    public void trimiteScrisoare(String dest)
    {
        System.out.println("Scrisoarea a fost trimisa" +
                " la destinatia: " + catre(dest));
    }

    /** Programul principal.*/
    public static void main(String[] args)
    {
        Scrisoare s = new Scrisoare();

        //trimite scrisoarea la destinatie
        s.trimiteScrisoare("Romania");

    }
}
```

Este evident faptul că aplicaţia nu are o utilitate practică, din moment ce afişează chiar stringul pe care îl primeşte ca parametru. Ea reprezintă doar un pretext pentru a vedea cum se crează o clasă interioară locală. Este uşor de observat că metoda `catre` cuprinde clasa interioară `Destinatie` în interiorul ei. Fiind definită în această metodă, clasa `Destinatie` este accesibilă doar în cadrul metodei. Încercarea de a o utiliza în afara metodei respective produce o eroare la compilarea programului.

Prin rularea programului se obţine următorul rezultat:

```
Scrisoarea a fost trimisa la destinatia: Romania
```

Pentru a complica şi mai mult lucrurile, trebuie să adăugăm că o clasă interioară poate fi definită şi în cadrul unui bloc de instrucţiuni, asemănător cu exemplul anterior, în care metoda `catre` este modificată astfel:

Digitized by Google

```
public String catre(String dest)
{
    if(dest != null)
    {
        /**
         * Clasa interioara Destinatie.
         */
        class Destinatie
        {
            /** Destinatia unei scrisori.*/
            private String dest;


            /** Creaza o destinatie pentru scrisoare.*/
            Destinatie(String dest)
            {
                this.dest = dest;
            }

            /** Obtine destinatia scrisorii.*/
            public String obtineDestinatia()
            {
                return dest;
            }
        }

        return (new Destinatie(dest)).obtineDestinatia();
    }
    else
    {
        return null;
    }
}
```

Deşi această utilizare pare destul de curioasă, clasa interioară `Destinatie` poate fi definită în cadrul unui bloc de instrucţiuni (în cazul nostru, instrucţiunea `if`), fiind vizibilă doar în acel bloc. Pentru a risipi eventuale urme de îndoială este util de reţinut că definirea clasei `Destinatie` în cadrul instrucţiunii `if`, nu înseamnă că această clasă este creată în funcţie de îndeplinirea condiţiei din `if`. Clasa va fi creată ca orice altă clasă la compilarea programului, dar nu va putea fi utilizată decât în cadrul blocului de instrucţiuni în care a fost definită.

Clasele locale sunt foarte rar utilizate din cauza codului mai puţin lizibil pe care îl crează (noi nu le vom utiliza deloc în cadrul acestei lucrări). Totuşi ele există şi pot fi utilizate în situaţii în care programatorul consideră că este nevoie de ele.

Digitized by Google

5.7.5 Clase anonime

Dacă exemplele anterioare v-au creat impresia că noţiunea de clasă locală este mai "deosebită", pregătiţi-vă să vă întăriţi această convingere, urmărind cu atenţie modul de definire al claselor anonime. Pentru o mai bună înţelegere, iată un exemplu ce defineşte o clasă anonimă:

Listing 5.18: Model de definire a unei clase anonime

```
1 /**
2  * Clasa exterioara Scrisoare
3  */
4 public class Scrisoare
5 {
6     public Destinatie trimite()
7     {
8         return new Destinatie()
9         {
10            private String dest2;
11
12            public String obtineDestinatia()
13            {
14                return "Germania";
15            }
16        }; //";" este necesar la definirea claselor anonime
17    }
18
19    /** Programul principal.*/
20    public static void main(String[] args)
21    {
22        Scrisoare s = new Scrisoare();
23
24        Destinatie d = s.trimite();
25    }
26 }
27
28 class Destinatie
29 {
30     private String dest;
31 }
```

Analizând exemplul întâlnim două clase, `Scrisoare` şi `Destinatie`, şi o modalitate mai puţin întâlnită de definire a metodei `trimite`. Această metodă combină procesul de returnare a unei instanţe a clasei `Destinatie`, la linia 8, cu cel de definire a unei clase fără nume (anonimă), la liniile 9-16. Clasa anonimă reprezintă o subclasă a clasei `Destinatie`, datorită faptului că definirea clasei apare imediat după numele clasei `Destinatie`. În consecinţă mai apare o clasă, despre care nu putem spune ce nume are, ci doar că extinde

Digitized by Google

clasa `Destinatie`.

Definirea metodei `trimite` poate fi analizată în două părţi:

- crearea şi returnarea unei instanţe a clasei `Destinatie`:

```
return new Destinatie()
```

- definirea clasei anonime:

```
private String dest2;

public String obtineDestinatia()
{
    return "Germania";
}
```

Această construcţie trebuie privită ca o modalitate de a defini o clasă care derivează clasa `Destinatie`, fără a mai specifica numele acestei clase. Mai precis, clasa anonimă este prescurtarea codului următor:

```
class DestinatiaMea extends Destinatia
{
    private String dest2;

    public String obtineDestinatia()
    {
        return "Germania";
    }
}

return new DestinatiaMea();
```

Una dintre cele mai importante caracteristici ale claselor interioare anonime este aceea că nu pot avea constructori expliciţi. Este evidentă această concluzie din moment ce o astfel de clasă nu are un nume.

Clasa de bază `Destinatie` este însă o clasă exterioară (deci, neanonimă), care poate avea constructori de orice tip. Apelul unui constructor neimplicit (cu parametri) se poate realiza astfel:

```
public class Scrisoare
{
    ...

    public Destinatie trimite(String destinatie)
    {
        /** Utilizarea constructorului explicit.*/
        return new Destinatie(destinatie)
        {
```


Digitized by Google

```
            private String dest2;

            public String obtineDestinatia()
            {
                return "";
            }
        };
    }

    ...
}

class Destinatie
{
    private String dest;

    Destinatie(String dest)
    {
        this.dest = dest;
    }
}
```

Nici un exemplu de utilizare a claselor anonime, dintre cele prezentate până în acest moment, nu poate să execute operaţii de iniţializare asupra atributelor clasei anonime, pentru a simula funcţionalitatea unui constructor explicit. Pentru a realiza acest lucru este necesar ca parametrul cu care va fi iniţializat atributul în cauză să fie declarat `final`, ca în exemplul:

```
//argumentul este declarat final.
public Destinatie trimite(final String destinatie)
{
    return new Destinatie()
    {
        private String dest2 = destinatie;

        public String obtineDestinatia()
        {
            return dest2;
        }
    };
}
```

Aşa cum aţi putut constata deja, clasele anonime crează un cod mai puţin lizibil, motiv pentru care este indicat ca folosirea lor să fie limitată la acele clase care sunt foarte mici ca dimensiune (o metodă sau două) şi care au o utilizare foarte uşor de înţeles. Veţi fi probabil surprinşi să aflaţi că probabil cel mai des utilizate clase interioare în practică sunt cele anonime. Clasele anonime sunt preferate de programatori când este vorba de a defini clase simple derivate din

Digitized by Google

`Thread` (vezi capitolul 8), clase de tip `Listener` (utilizate foarte frecvent în `AWT` şi `Swing`) sau alte clase având o singură metodă, deoarece codul clasei apare exact în locul în care este folosit, uşurând astfel înţelegerea programului. Clasele anonime mai au şi avantajul că se pot aduce mici modificări claselor existente fără a încărca inutil aplicaţia cu o arborescenţă inutilă de clase triviale.

5.7.6 Avantajele şi dezavantajele claselor interioare

Avantajele utilizării claselor interioare

Deşi par incomode la prima vedere, clasele interioare nu reprezintă doar un simplu mecanism de "ascundere" a codului, prin plasarea unor clase în interiorul altor clase. Clasele interioare sunt capabile de mai mult, pentru că au posibilitatea de comunicare cu clasa exterioară.

În principiu, clasele interioare oferă următoarele tipuri de avantaje:

- avantajul orientării pe obiecte

  Folosind clasele interioare codul devine chiar mai orientat pe obiecte, decât în lipsa lor, deoarece ele permit "eliminarea" anumitor părţi de cod din cadrul unei clase (clasa exterioară) şi introducerea lor într–o clasă proprie (clasa interioară). Altfel spus, din punct de vedere al orientării pe obiecte, funcţionalitatea care nu aparţine de clasa exterioară este înlăturată şi plasată într-o clasă interioară, rezultând astfel o decuplare a funcţionalităţilor clasei exterioare şi un cod mult mai elegant şi mai clar. Mai mult, clasa interioară poate avea în continuare acces la toate atributele şi metodele clasei exterioare, în cazul în care este membru nestatic al clasei exterioare;

- avantajul organizării claselor

  Clasele interioare permit o organizare mai bună a structurii pachetelor. Astfel, clasele care nu trebuie să fie accesibile altor clase, chiar şi celor din acelaşi pachet, pot fi definite ca fiind clase interioare, în aşa fel încât să fie accesate doar de clasa exterioară care le conţine.

Dezavantajele utilizării claselor interioare

Clasele interioare prezintă dezavantaje ce nu trebuie neglijate. Din punctul de vedere al întreţinerii aplicaţiilor, programatorii Java neexperimentaţi pot considera clasele interioare foarte dificil de înţeles. Este o reacţie normală deoarece este necesară o perioadă de timp pentru ca programatorii Java să se obişnuiască cu acest concept şi să considere utilă folosirea lui. Un alt dezavantaj ce merită a

Digitized by Google

fi semnalat este că utilizarea claselor interioare produce o creştere a numărului de clase ale aplicaţiei.

Concluzii

Clasele interioare reprezintă un concept sofisticat care este întâlnit în mai multe limbaje de programare obiect-orientate. Nivelul de complexitate mai ridicat decât al altor concepte ale limbajului Java, face ca acest tip de clase să fie mai puţin preferat de programatorii începători. Cu timpul însă, ei acumulează experienţa necesară pentru a recunoaşte situaţiile în care se impune utilizarea claselor interioare. Din acest motiv, pentru cei aflaţi în faza de iniţiere în limbajul Java este suficient dacă reuşesc să se familiarizeze cu sintaxa şi modul lor de utilizare.

5.8 Identificarea tipurilor de date în faza de execuţie

Pentru fiecare obiect existent în cadrul unei aplicaţii, limbajul Java oferă posibilitatea de a afla informaţii cu privire la clasa a cărei instanţă este acel obiect. Aceste informaţii pot fi colectate în faza de execuţie a aplicaţiei respective ("runtime").

Pentru a vă convinge că astfel de informaţii sunt necesare, vom considera un exemplu, care va arăta în mod explicit de ce uneori este nevoie să cunoaştem tipul de date al unui obiect în faza de execuţie a programului. Să presupunem că avem clasa de bază `Masina`, din care se derivează alte clase, câte una pentru fiecare tip de maşină. Astfel, avem clasele `Dacia`, `Opel`, `Ferrari`, etc. Să presupunem că toate maşinile au un preţ, şi că pentru fiecare maşină există un mod diferit de a-i calcula costul. În Java, aceasta înseamnă că metoda `afiseazaPret()` este o metodă abstractă a clasei de bază `Masina`, care are câte o implementare diferită pentru fiecare dintre clasele `Dacia`, `Opel`, `Ferrari`.

Listing 5.19: Clasa abstractă `Masina` şi clasele derivate din ea

```
//Continutul fisierului Identificare.java
import java.util.*;

abstract class Masina
{
  abstract void afiseazaPret();
}

```


Digitized by Google

```
class Dacia extends Masina
{
  void afiseazaPret()
  {
    System.out.println("Pretul unei Dacii este de " +
    "5.000 de dolari.");
  }
}

class Opel extends Masina
{
  void afiseazaPret()
  {
    System.out.println("Pretul unui Opel este de " +
    "15.000 de dolari.");
  }
}

class Ferrari extends Masina
{
  void afiseazaPret()
  {
    System.out.println("Pretul unui Ferrari este de " +
    "75.000 de dolari.");
  }
}

public class Identificare
{
  public static void main(String[] args)
  {
    Vector v = new Vector();

    v.add(new Dacia());
    v.add(new Opel());
    v.add(new Ferrari());

    for(int i = 0 ; i < v.size(); i++)
    {
      ((Masina) v.elementAt(i)).afiseazaPret();
    }
  }
}
```

După cum reiese din **Listing 5.19**, fiecare clasă care extinde clasa de bază `Masina` are propria ei definire a metodei `afiseazaPret()`. Metoda `main()` a programului principal crează un vector de elemente, în care se stochează câte un obiect, instanţă a unei clase derivate din clasa de bază `Masina`. Deoarece clasa `Vector` stochează doar instanţe ale clasei `Object`, fiecare dintre in-

Digitized by Google

stanţele anterioare va fi stocată ca un obiect de tip `Object`, pierzându-se (aparent) informaţia specifică tipului a cărei instanţă este de fapt. În final, vectorul este parcurs şi pentru fiecare element al său este apelată metoda `afiseazaPret()`. Operaţia de *cast* (conversia explicită la tipul `Masina`) este necesară pentru a putea apela metoda `afiseazaPret()`, care nu există în clasa `Object`, redând astfel o parte din funcţionalitatea obiectelor. Funcţionalitatea specifică nu este redată în totalitate deoarece în acest moment se cunoaşte doar faptul că vectorul conţine instanţe ale claselor derivate din clasa `Masina`, fără a cunoaşte exact aceste clase (`Dacia`, `Opel`, etc.). Totuşi, după cum am văzut în paragraful 5.2.3, apelul metodei `afiseazaPret()` pe un obiect convertit la clasa `Masina`, va conduce prin polimorfism la apelul metodei cu acelaşi nume din clasa care a fost utilizată la crearea instanţei (`Dacia`, `Opel`, sau `Ferrari`), obţinându-se următorul rezultat:

```
Pretul unei Dacii este de 5.000 de dolari.
Pretul unui Opel este de 15.000 de dolari.
Pretul unui Ferrari este de 75.000 de dolari.
```

Astfel, folosind polimorfismul se pot crea instanţe ale unor clase specifice (`Dacia`, `Opel`, `Ferrari`), care sunt convertite apoi la o clasă de bază (`Masina`), ceea ce implică pierderea tipului specific al obiectului, după care, pe parcursul programului, se pot utiliza aceste referinţe către clasa de bază. Efectul este că metoda exactă care va fi apelată pentru un obiect convertit la clasa `Masina` este determinată de referinţă, care poate fi către una din clasele: `Dacia`, `Opel`, `Ferrari`.

În general, exact acest lucru se urmăreşte: ca secvenţele de cod să deţină cât mai puţine informaţii despre tipul specific al unui obiect şi să utilizeze doar reprezentarea generală a unei familii de obiecte (în cazul nostru, maşini). Ceea ce rezultă este un cod sursă mult mai uşor de scris, citit, înţeles şi modificat. Este motivul pentru care polimorfismul este unul dintre principiile esenţiale ale programării orientate pe obiecte (şi totuşi, atât de neglijate de începători).

Există însă situaţii în care este utilă chiar cunoaşterea *exactă* a tipului unui obiect. De exemplu, dacă doriţi să specificaţi că o maşină Ferrari este disponibilă doar în culoarea roşie, este necesar să testăm obiectul respectiv, pentru a vedea ce clasă instanţiază.

În Java există două metode de a afla tipul de date al unui obiect:

- identificarea standard a tipului de date, ce presupune că toate tipurile de date sunt disponibile atât în faza de compilare cât şi în cea de execuţie a aplicaţiei;

Digitized by Google

- mecanismul de reflecţie, care descoperă informaţii despre o clasă doar în timpul fazei de execuţie.

5.8.1 Identificarea standard a tipurilor de date

În Java, identificarea standard a tipurilor de date se poate realiza în trei moduri:

- Utilizând operatorul de *cast* (în exemplul anterior acesta a fost utilizat la conversia către clasa `Masina`). Dacă operaţia de *cast* este incorectă (dacă obiectul nu poate fi covertit la tipul specificat), atunci va fi aruncată o excepţie de tipul `ClassCastException`;

- Utilizând un obiect de tip `Class`, care conţine informaţii despre clasa respectivă. Uneori clasa `Class` mai este denumită şi *metaclasă*. Maşina virtuală Java ataşează *fiecărei* clase existente într-un program un obiect de tipul `Class`, care este utilizat pentru a crea instanţe ale clasei respective, dar care poate fi utilizat şi de programator pentru a afla date despre acea clasă. Există două posibilităţi de a obţine obiectul de tip `Class` asociat unei clase (pentru exemplificare am ales clasa `Identificare` din programul anterior):

  – folosind metoda `forName` a clasei `Class`:

```
Class c = null;
try
{
    c = Class.forName("Identificare");
}
catch(ClassNotFoundException cnfe)
{
}
```

  – folosind atributul `.class` asociat oricărei clase:

```
Class c = Identificare.class;
```

  În general se preferă utilizarea celei de-a doua modalităţi, din mai multe motive. În primul rând este mai simplă, deoarece nu necesită prinderea şi tratarea excepţiei `ClassNotFoundException`. În al doilea rând este mai sigură, deoarece existenţa clasei `Identificare` este verificată în faza de compilare şi nu în cea de execuţie, ca în primul caz.

Digitized by Google

Obiectul de tip `Class` asociat unei clase poate fi obţinut şi cu ajutorul unei instanţe a clasei respective, utilizând metoda `getClass()` a clasei `Object`:

```
Masina m = new Dacia();
Class c = m.getClass();
```

- Utilizând cuvântul cheie `instanceof`, care verifică dacă un obiect este instanţă a unei clase specificate şi returnează o valoare booleană. Iată un exemplu de utilizare în contextul aplicaţiei anterioare:

```
Masina m = new Dacia();

if (m instanceof Dacia)
{
    System.out.println("Masina aceasta este o Dacie.");
}
else
{
    System.out.println("Masina aceasta NU este o Dacie.");
}
```

Rezultatul execuţiei acestei secvenţe este:

```
Masina aceasta este o Dacie.
```

Limbajul Java oferă o alternativă pentru operatorul `instanceof`: metoda `isInstance()` a clasei `Class`. Această metodă are un avantaj faţă de operatorul `instanceof`: poate fi utilizată şi pe obiecte de tipul `Class`. De exemplu, este greşită o secvenţă de cod de tipul următor:

```
Class[] c = {Dacia.class, Opel.class, Ferrari.class};

Masina m = new Dacia();

for (int i = 0; i < c.length; i++)
{
    if (m instanceof c[i]) //EROARE
    {
        System.out.println("m este instanta a clasei " +
                           c[i].getName());
    }
}
```

Eroarea este datorată faptului că operatorul `instanceof` nu poate fi folosit decât în conjuncţie cu numele clasei (`Dacia`), şi nu cu un obiect de tipul `Class` (`Dacia.class`). Pentru a corecta această eroare, se

Digitized by Google

utilizează metoda `isInstance()`, înlocuind condiţia din instrucţiunea `if`, cu condiţia `c[i].isInstance(m)`. Rezultatul execuţiei programului astfel modificat este:

```
m este instanta a clasei Dacia
```

După cum se poate observa, metoda `isInstance()` oferă un mod dinamic de a apela operatorul `instanceof`. Ambele variante produc însă rezultate echivalente.

Există o diferenţă importantă între operatorul `instanceof` (implicit şi metoda `isInstance()`, pentru că sunt operaţii cu rezultat echivalent) şi compararea folosind obiecte de tipul `Class`. Diferenţa este vizibilă la aflarea informaţiilor despre clase. Operatorul `instanceof` verifică dacă un obiect este o instanţă a unei clase specificate *sau* a unei clase derivate din clasa specificată, în timp ce compararea obiectelor de tipul `Class` nu ţine cont de moştenire, ci doar verifică strict dacă obiectul respectiv este o instanţă a clasei specificate sau nu. Pentru a verifica afirmaţiile anterioare, iată un exemplu:

```
Masina m = new Dacia();

System.out.println("m instanceof Masina " +
                   (m instanceof Masina));
System.out.println("m instanceof Dacia " +
                   (m instanceof Dacia));

System.out.println("m.getClass() == Masina.class " +
                   (m.getClass() == Masina.class));
System.out.println("m.getClass() == Dacia.class " +
                   (m.getClass() == Dacia.class));
```

Secvenţa afişează următorul rezultat:

```
m instanceof Masina true
m instanceof Dacia true
m.getClass() == Masina.class false
m.getClass() == Dacia.class true
```

5.8.2 Mecanismul de reflecţie ("reflection")

Acest paragraf prezintă concepte mai avansate, care nu sunt necesare pentru înţelegerea informaţiei din restul lucrării. Dacă nu sunteţi direct interesaţi de

Digitized by Google

conceptul de reflection, puteţi sări deocamdată peste el şi veţi putea reveni ulterior, când bagajul de cunoştinţe acumulat vă va permite o înţelegere mai uşoară a lui.

Pentru a putea afla tipul unui anumit obiect puteţi utiliza oricare dintre metodele anterioare, dar trebuie să aveţi în vedere că există o anumită limitare. Dezavantajul constă în faptul că tipurile de date trebuie să fie cunoscute în faza de compilare, chiar dacă ele sunt utilizate în faza de execuţie. Cu alte cuvinte, compilatorul trebuie să cunoască toate clasele care sunt utilizate pentru identificarea tipurilor de date în faza de execuţie a programului.

La prima vedere nu pare a fi o limitare, dar poate deveni în situaţia în care este necesară instanţierea unei clase care nu este disponibilă maşinii virtuale JVM în momentul compilării. Aceasta sună probabil în acest moment a science-fiction, dar în aplicaţiile concrete (în special cele client-server) această situaţie nu constituie o raritate. De exemplu, într-un fişier putem avea un şir de octeţi care ştim că reprezintă o clasă. În faza de compilare, compilatorul nu poate afla nici o informaţie legată de acel şir de octeţi, deci utilizarea unei astfel de clase pare imposibilă. Din fericire, platforma Java oferă o soluţie pentru această problemă: mecanismul de reflecţie (*reflection*). Reflecţia este un mecanism prin care se pot determina informaţii despre o clasă (numele clasei, numele metodelor, numele constructorilor, etc.) fără ca acea clasă să fie cunoscută compilatorului în faza de compilare.

Clasa `Class` prezentată în paragraful anterior suportă conceptul de reflecţie, dar majoritatea claselor specializate pentru acest mecanism se găsesc în pachetul `java.lang.reflect`. Câteva dintre aceste clase, `Constructor`, `Field`, `Method`, vor fi prezentate mai pe larg pe parcursul acestui subcapitol.

În general, mecanismul de reflecţie este utilizat în mod direct destul de rar de programatori, dar există situaţii în care sunt de folos anumite informaţii legate de o clasă. De aceea vom prezenta, cu ajutorul unor exemple simple, cum se pot afla informaţii utile despre o clasă.

Mediul de programare Java are la dispoziţie un API destul de bogat pentru a asigura funcţionalitatea mecanismului de reflecţie. API-ul reprezintă clasele, obiectele şi interfeţele din maşina virtuală JVM şi permite operaţii de tipul:

- determinarea clasei a cărei instanţă este un anumit obiect;
- obţinerea de informaţii despre modificatorii de acces ai clasei, despre atribute, metode, constructori şi superclase;
- obţinerea de informaţii despre constantele şi metodele declarate într-o interfaţă;

Digitized by Google

- instanţierea unei clase, al cărei nume este cunoscut doar în faza de execuţie;
- modificarea valorii unui atribut al unei clase, chiar dacă numele atributului este cunoscut doar în faza de execuţie;
- apelarea unei metode a unei clase, chiar dacă numele acestei metode este cunoscut doar în faza de execuţie.

Obţinerea de informaţii despre o clasă

Pentru a afla informaţii despre o clasă trebuie să obţinem obiectul de tipul `Class` asociat clasei respective, pentru că acel obiect deţine toate informaţiile legate de clasă. Folosind acest obiect se pot apela diverse metode ale clasei `Class`, care returnează obiecte de tipul `Constructor`, `Field`, `Method`, corespunzătoare constructorilor, atributelor şi respectiv metodelor definite în cadrul clasei. De asemenea, un obiect `Class` poate reprezenta chiar şi o interfaţă, situaţie în care se pot afla informaţii despre constantele definite, metodele declarate etc. Evident, nu toate metodele clasei `Class` sunt potrivite într-o astfel de situaţie. De exemplu, nu este normal apelul metodei `getConstructors`, deoarece o interfaţă nu are constructori.

Aşadar, primul pas care trebuie realizat pentru o obţine informaţii despre o clasă este obţinerea obiectului de tipul `Class` asociat clasei respective. Principalele metode de a realiza acest demers au fost prezentate de-a lungul acestui subcapitol, dar este util să le reamintim:

- dacă există o instanţă a clasei respective, se utilizează metoda `getClass`, prezentă în orice clasă Java (care extinde clasa `Object`):

```
Ferrari f = new Ferrari();
Class c = f.getClass();
```

- dacă numele clasei este disponibil în faza de compilare, se utilizează sufixul `.class` ataşat numelui clasei:

```
Class c = Ferrari.class;
```

- dacă numele clasei nu este disponibil în faza de compilare, dar devine disponibil în faza de execuţie, atunci se foloseşte metoda `forName` a clasei `Class`:

```
Class c = Class.forName("Ferrari");
```


Digitized by Google

Dacă obiectul de tip `Class` a fost obţinut, pe baza lui se pot determina alte informaţii utile legate de clasa respectivă:

- Numele clasei, utilizând o secvenţă de cod asemănătoare cu:

```
Ferrari f = new Ferrari();
Class c = f.getClass();

//determina numele clasei
String name = c.getName();

//afiseaza acest nume
System.out.println(name);
```

  Execuţia secvenţei anterioare are ca rezultat afişarea stringului `"Ferrari"`.

- Tipul de modificator al clasei (`public`, `protected`, `abstract`, `final`, etc.)[4]:

```
Ferrari f = new Ferrari();
Class c = f.getClass();

int m = c.getModifiers();

if (Modifier.isPublic(m))
{
    System.out.println("public");
}

if (Modifier.isFinal(m))
{
    System.out.println("final");
}
```

  Secvenţa nu va afişa nimic, deoarece clasa `Ferrari` nu este declarată nici `public`, nici `final`.

- Obiectul de tip `Class` asociat clasei de bază din care a fost derivată clasa respectivă (dacă este cazul):

```
Ferrari f = new Ferrari();
Class c = f.getClass();
Class sc = c.getSuperclass();
System.out.println(sc.getName());
```

[4] Pentru ca secvenţa sa funcţioneze trebuie importate clasele din pachetul `java.lang.reflect`. Analog şi pentru restul exemplelor.

Digitized by Google

Secvenţa va afişa stringul "Masina".

- Obiectele de tip `Class` asociate interfeţelor pe care le implementează clasa respectivă (dacă este cazul):

```
...
Class[] ci = c.getInterfaces();
...
```

- Dacă este clasă sau interfaţă:

```
Class c;
...
if (c.isInterface())
{
    ...
}
...
```

- Informaţii despre atributele clasei, utilizând metoda `getFields()` a clasei `Class` (se pot determina numele, tipul, modificatorii de acces ai fiecărui atribut, sau chiar se pot da valori acestor atribute):

```
Class c = Ferrari.class;
Field[] fields = c.getFields();
for (int i = 0; i < fields.length; i++)
{
    String fieldName = fields[i].getName();
    String fieldType = (fields[i].getType()).getName();

    System.out.println(fieldName + " " + fieldType);
}
```

- Informaţii despre constructorii unei clase, utilizând metoda `getConstructors()` a clasei `Class` (se pot determina numele, lista parametrilor constructorului, etc.):

```
Class c = Ferrari.class;
Constructor[] constr = c.getConstructors();
for (int i = 0; i < constr.length; i++)
{
    String name = constr[i].getName();
    ...
}
...
```


Digitized by Google

- Informaţii despre metodele clasei, utilizând metoda `getMethods()` a clasei `Class` (se pot determina numele, tipul valorii returnate, lista argumentelor metodei, sau se poate apela metoda în sine folosind metoda `invoke()` a clasei `Method`):

```
Class c = Ferrari.class;
Method[] methods = c.getMethods();
for (int i = 0; i < methods.length; i++)
{
    String name = methods[i].getName();
    ...
}
...
```

Obiectul de tip `Class` poate fi utilizat nu numai pentru a interoga clasele şi a afla diverse informaţii despre ele. Cu ajutorul lui se pot realiza şi instanţieri ale claselor, modificări ale atributelor, sau apelări ale metodelor. Iată câteva exemple de acest tip:

- Crearea unei instanţe a unei clase se poate realiza prin intermediul metodei `newInstance()` a clasei `Class`, în două moduri, în funcţie de numărul de argumente al constructorului:

  – constructori fără argumente

```
Object o = null;
try
{
    Class c = Class.forName("Ferrari");
    o = c.newInstance();
}
catch(Exception e)
{
}
```

  – constructori cu argumente

```
try
{
    Class[] argsClasses = {String.class};
    Object[] argsValues = {"12"};

    Class c = Class.forName("java.lang.Integer");
    Constructor constr = c.getConstructor(argsClasses);

```


Digitized by Google

```
    Object i = constr.newInstance(argsValues);

    System.out.println(i);
}
catch(Exception e)
{
}
```

Pentru o mai bună înţelegere a instanţierii prin *reflection* a unei clase cu ajutorul unui constructor cu parametri, vom prezenta mai amănunţit acest exemplu. Instrucţiunile folosite au ca rezultat crearea unui obiect de tip `Integer`, utilizând un constructor al acestei clase, care are ca parametru un obiect `String`. În esenţă, codul este echivalent cu secvenţa:

```
String s = "12";
Integer i = new Integer(s);
```

Pentru a realiza acest lucru, au fost create mai întâi şirul cu tipurile argumentelor (`String`) şi şirul cu valorile argumentelor (`"12"`). Apoi, pornind de la obiectul `Class` ataşat clasei `Integer`, s–a obţinut acel constructor al clasei `Integer` care primeşte ca parametru un obiect `String`. În final, utilizând acest constructor s–a creat o instanţă a clasei `Integer`, cu argumentele specificate. În urma execuţiei se afişează valoarea `"12"`.

O diferenţă importantă faţă de varianta apelului constructorului fără parametri este aceea că `newInstance()` aparţine clasei `Constructor` şi nu clasei `Class` ca în exemplul precedent.

- Obţinerea valorii unui atribut dintr-o clasă se realizează cu ajutorul metodei `getField()`. Aceasta returnează un obiect de tipul `Field`, care conţine datele despre atributul respectiv. Dacă tipul de dată al atributului este primitiv, atunci valoarea atributului se obţine apelând una din metodele `getInt()`, `getFloat()`, `getBoolean()`, etc., corespunzător tipului primitiv de dată al atributului. Dacă atributul reprezintă un obiect, atunci trebuie utilizată metoda `get()` pentru a afla valoarea acestuia. Drept exemplu vom prezenta o secvenţă de cod ce obţine valoarea unui atribut (pentru aceasta presupunem că avem definită o clasă `Audi`, ce conţine atributul `model` de tip `String`):

```
try
{
    Audi a = new Audi("TT–Coupe");
    Class c = a.getClass();
```


Digitized by Google

```

    //obtine obiectul ce contine informatii despre model
    Field f = c.getField("model");
    //obtine valoarea atributului model pentru instanta a
    String s = (String) f.get(a);

    System.out.println("Model = " + s);
}
catch(Exception e)
{
}
```

- Similar metodelor `get()` (`getInt()`, `getFloat()`, `get()` etc.), clasa `Field` mai cuprinde şi metode `set` (`setInt()`, `setFloat()`, `set()` etc.) pentru a putea modifica valoarea atributului, setându–o la noua valoare dorită (pentru simplitate vom considera aceleaşi ipoteze ca la exemplul anterior):

```
try
{
    Audi a = new Audi("TT-Coupe");
    Class c = a.getClass();

    //obtine obiectul ce contine informatii despre model
    Field f = c.getField("model");

    //modificam valoarea atributului model in "A4"
    //pentru instanta a
    f.set(a, "A4");
}
catch(Exception e)
{
}
```

- Apelul unei metode oarecare a unei clase oarecare se poate realiza dinamic prin intermediul clasei `Method`. De exemplu, pentru a concatena două stringuri se poate utiliza metoda `concat()` a clasei `String`, ceea ce ar conduce, folosind apelul dinamic al metodei respective, la următoarea secvenţă de cod ce afişează în final mesajul `"Limbajul Java"`:

```
try
{
    //obtinerea obiectului Class asociat clasei String
    Class c = String.class;

    //definire tipuri si valori ptr. parametrii metodei
    //ce va fi apelata
    Class[] tipParam = {String.class};
```


Digitized by Google

```
    Object[] valParam = {"Java"};

    //obtine obiectul Method ce contine informatii legate
    //de metoda concat() cu lista de parametri specificata
    Method m = c.getMethod("concat", tipParam);

    //apel dinamic metoda concat() pe obiectul "Limbajul "
    //echivalent cu: "Limbajul ".concat("Java");
    String s = (String) m.invoke("Limbajul ", valParam);

    //afisarea rezultatului concatenarii celor 2 siruri
    System.out.println(s);
}
catch(Exception e)
{
}
```

Exemplele prezentate de-a lungul acestui subcapitol denotă faptul că mecanismul de reflecţie permite ca informaţiile despre obiecte să fie complet determinate în faza de execuţie, fără nici o intervenţie în faza de compilare. Analizând comparativ identificarea standard a tipului de date şi mecanismul de reflecţie se poate spune că cea mai importantă diferenţă dintre cele două este că în primul caz compilatorul verifică în faza de compilare clasele, în timp ce în cel de–al doilea caz, clasele sunt verificate în timpul fazei de execuţie.

Rezumat

Moştenirea este o caracteristică puternică, fiind esenţa programării orientate pe obiecte şi a limbajului Java. Ea permite abstractizarea funcţionalităţii în clase abstracte, pentru a deriva apoi din aceste clase, alte clase care implementează şi măresc funcţionalitatea de bază.

Cea mai abstractă clasă, care nu implementează nimic poate fi specificată utilizând o *interfaţă*. Metodele specificate de o interfaţă trebuie definite de clasa care o implementează.

Un alt tip special de clase sunt clasele interioare, care sunt clase definite în interiorul altor clase. Deşi mai puţin utilizate decât celelalte tipuri de clase, ele oferă facilităţi specifice şi au avantaje importante.

Uneori este necesar să identificăm tipul unui obiect în faza de execuţie a programului. Deşi această facilitate nu va fi utilizată în cadrul acestei lucrări, ea reprezintă un element *sine qua non* pentru programatorii care ajung să folosească tehnologii client-server cum ar fi CORBA sau EJB.

Digitized by Google

Capitolul următor este unul mult aşteptat, prin prisma numărului de referiri de care a avut parte de-a lungul capitolelor anterioare. Este vorba despre "Tratarea excepţiilor".

Noţiuni fundamentale

cast: operator unar prin care o clasă este convertită la altă clasă.

clasă abstractă: clasă care nu poate fi instanţiată, dar care înglobează funcţionalitatea comună a claselor derivate din ea.

clasă anonimă: clasă locală care nu are nume.

clasă de bază: clasă din care se derivează/implementează alte clase.

clasă finală: clasă care nu mai poate fi derivată.

clasă interioară: clasă definită în interiorul altei clase.

clasă derivată: clasă complet nouă care moşteneşte funcţionalitatea clasei de bază.

clasă locală: clasă interioară definită în interiorul unui bloc de cod.

clasă wrapper: clasă care oferă un obiect ce stochează un tip primitiv. De exemplu, `Integer` este o clasă wrapper peste tipul primitiv `int`.

`Class`: clasă specială ce conţine informaţii despre o clasă Java.

constructor `super`: constructorul clasei de bază din care a fost derivată clasa de faţă.

`extends`: cuvânt cheie prin care se specifică faptul că o clasă este derivată din altă clasă.

`implements`: cuvânt cheie prin care se specifică faptul că o clasă implementează metodele expuse de o interfaţă.

`instanceof`: operator prin care se verifică dacă un obiect este o instanţă a unei clase specificate.

interfaţă: tip special de clasă Java, care nu conţine nici un detaliu de implementare.

mecanismul "reflection": permite obţinerea de informaţii despre o clasă în timpul fazei de execuţie a programului.

moştenire: proces prin care se poate deriva o clasă nouă dintr-o clasă de bază, fără a afecta implementarea clasei de bază. Prin aceasta se poate crea o ierarhie de clase.

moştenire multiplă: proces prin care o clasă este derivată din mai multe clase de bază. Acest lucru nu este permis în Java. Totuşi există o alternativă: implementarea mai multor interfeţe.

Digitized by Google

polimorfism: abilitatea unei variabile referinţă de a referi obiecte de tipuri diferite. Când se apelează o metodă pe această variabilă, se apelează de fapt metoda tipului de obiect referit în momentul respectiv.

subclasă: denumire a clasei derivate.

superclasă: denumire a clasei de bază.

Erori frecvente

1. Membrii `private` din clasa de bază nu sunt vizibili în clasa derivată.

2. Dacă nu se defineşte nici un constructor în clasa derivată, iar clasa de bază nu are constructor implicit, se obţine o eroare la compilare.

3. Clasele abstracte nu pot fi instanţiate.

4. Dacă în clasa derivată uităm să implementăm o metodă definită abstractă în clasa de bază, atunci clasa derivată devine ea însăşi abstractă, ca şi superclasa ei. Aceasta va genera o eroare la compilare (evident, dacă nu declarăm clasa derivată ca fiind abstractă). Clasa derivată devine concretă doar în momentul în care a implementat metoda abstractă moştenită.

5. Metodele finale nu pot fi redefinite, iar clasele finale nu pot fi extinse.

6. Metodele statice folosesc legarea statică, chiar dacă sunt redefinite în clasele derivate.

7. În clasa derivată atributele moştenite de la clasa de bază ar trebui iniţializate ca un agregat, folosind metoda `super`. Dacă aceşti membri sunt `public` sau `protected`, ei vor putea fi ulterior modificaţi separat.

8. Dacă o metodă generică întoarce o referinţă generică, atunci de obicei trebuie realizată o conversie de tip pentru a obţine obiectul efectiv returnat.

9. Membrii claselor interioare anonime pot fi iniţializaţi doar cu atribute `final` ale clasei exterioare.

Digitized by Google

Exerciţii

Pe scurt

1. Care membri din clasa de bază pot fi folosiţi în clasa derivată? Care membri devin publici pentru utilizatorii claselor derivate?

2. Ce este agregarea?

3. Explicaţi polimorfismul.

4. Explicaţi legarea dinamică. Când nu se foloseşte legarea dinamică?

5. Ce este o metodă `final`?

6. Care este diferenţa dintre o clasă finală şi alte clase? Când se folosesc clasele `final`?

7. Ce este o metodă abstractă?

8. Ce este o clasă abstractă?

9. Ce este o interfaţă? Prin ce diferă o interfaţă de o clasă abstractă? Ce fel de membri poate conţine o interfaţă?

10. Cum sunt implementate componentele generice în Java?

11. Ce este o clasă interioară şi de câte tipuri sunt clasele interioare?

12. Prezentaţi pe scurt procesul de identificare a tipurilor de date în timpul fazei de execuţie a programului.

În practică

1. Scrieţi două metode generice `min()` şi `max()`, fiecare acceptând doi parametri de tip `Comparable`. Folosiţi metodele într-o clasă pe care o denumiţi `MyInteger`.

2. Scrieţi două metode generice `min()` şi `max()`, fiecare acceptând un şir de `Comparable`. Folosiţi apoi aceste metode pentru tipul `MyInteger`.

3. Adăugaţi a nouă clasă (de exemplu, `Triunghi`) la ierarhia de clase Shape şi verificaţi prin intermediul metodei de sortare dacă polimorfismul funcţionează ca şi în cazul claselor celelalte.

Digitized by Google

4. Creaţi o ierarhie de clase `Felina`, care să conţină clasele `Pisica`, `Tigru`, `Leu`, `Leopard`, `Ghepard` etc. (pentru detalii consultaţi un atlas biologic...). Definiţi în clasa de bază metode care sunt comune pentru toate felinele, cum ar fi `getNume()`, `getCuloare()`, şi în fiecare clasă metode specifice fiecărei feline (de exemplu `toarce()`, sau `getSoarece()` în cazul clasei `Pisica`). Redefiniţi unele din metodele clasei de bază în clasele derivate. Creaţi un şir de feline şi apelaţi metodele din clasa de bază pentru a vedea ce se întâmplă.

5. Modificaţi exemplul anterior astfel încât clasa `Felina` sa fie o clasă abstractă. Definiţi metodele clasei `Felina` ca fiind abstracte ori de câte ori este posibil.

6. Creaţi o clasă abstractă fără nici o metodă abstractă şi verificaţi faptul că nu o puteţi instanţia.

7. Modificaţi exerciţiul 4 pentru a ilustra ordinea de construire a claselor de bază şi a celor derivate prin afişarea de mesaje în constructori. Apoi adăugaţi membri de tip referinţă la fiecare clasă şi urmăriţi ordinea în care sunt iniţializaţi aceşti membri.

8. Creaţi o clasă de bază cu două metode, astfel încât prima metodă să o apeleze pe cea de-a doua. Derivaţi o clasă şi redefiniţi cea de-a doua metodă. Creaţi acum o instanţă a clasei derivate, convertiţi-o folosind operatorul de cast la clasa de bază şi apelaţi prima metodă. Explicaţi rezultatul.

9. Modificaţi clasa `Shape` astfel încât ea să poată fi folosită de către un algoritm de sortare generic.

Proiecte de programare

1. Rescrieţi ierarhia de clase `Shape` pentru a reţine aria ca un membru privat, care este calculat de către constructorul clasei `Shape`. Constructorii din clasele derivate trebuie să calculeze aria şi să trimită rezultatul către metoda `super`. Faceţi din `area()` o metodă finală care doar returnează valoarea acestui atribut.

2. Adaugaţi conceptul de poziţie la ierarhia `Shape` prin includerea coordonatelor ca date membru. Adaugaţi apoi o metodă `distance`.

3. Scrieţi o clasă abstractă `Date` din care să derivaţi clasa `GregorianDate` (care reprezintă o dată în formatul nostru obişnuit).

Digitized by Google

6. Tratarea excepţiilor

> Dacă e verde, e biologie, dacă miroase urât e chimie, dacă are numere e matematică, dacă nu funcţionează e tehnologie.
>
> Autor necunoscut

> Nici o cantitate de experimente nu îmi poate valida teoria. Un singur experiment este însă suficient pentru a o infirma.
>
> Albert Einstein

În capitolul anterior am prezentat concepte importante ale programării orientate pe obiecte, cum ar fi moştenirea, polimorfismul, programarea generică şi clasele abstracte. În acest capitol vom prezenta modul în care se pot trata elegant erorile care apar inevitabil în timpul execuţiei unui program folosind din plin principiile prezentate până acum.

Vom afla în curând:

- Ce constituie o situaţie de eroare în Java;
- Ce sunt excepţiile şi de câte tipuri sunt ele;
- Care este ierarhia de excepţii în limbajul Java;
- Cum sunt folosite excepţiile pentru a semnala situaţii de eroare.

6.1 Ce sunt excepţiile?

Programatorii cunosc faptul că erori apar în orice aplicaţie software, indiferent de limbajul în care aceasta a fost scrisă. Dar ce se întâmplă *după* ce aceste

Digitized by Google

erori apar? Este posibil ca programul să le trateze şi să îşi reia mersul normal al execuţiei sau este pur şi simplu forţat să se termine? Dacă este posibilă tratarea lor, atunci cum este ea realizată şi de cine? Capitolul de faţă încearcă să răspundă tuturor acestor întrebări.

Java foloseşte noţiunea de excepţie pentru a oferi posibilitatea de tratare a erorilor pentru programele scrise în acest limbaj. Excepţia poate fi considerată ca un eveniment care are loc în timpul execuţiei unui program, prin care este afectată (întreruptă) secvenţa de execuţie normală a instrucţiunilor din acel program. Excepţiile reprezintă modul în care Java indică o situaţie anormală (de eroare) semnalată în procesul de execuţie a unei metode apelate.

Ca în orice alt limbaj de programare, în Java, erorile pot apare în ambele faze ale implementării unei aplicaţii software: compilare şi execuţie. Momentul ideal pentru a descoperi erorile dintr-un program este cel al compilării, înainte de a încerca execuţia programului. Totuşi, nu toate erorile pot fi descoperite în faza de compilare. Unele dintre ele nu pot fi detectate şi, implicit, tratate decât în faza de execuţie a programului (*run-time*). În Java, tratarea erorilor în faza de execuţie a unui program este posibilă prin intermediul excepţiilor.

Există numeroase tipuri de erori care pot provoca situaţii de excepţie, împiedicând astfel execuţia normală a unui program. Aceste probleme pot varia de la defecţiuni serioase de *hardware*, cum ar fi distrugerea parţială a unui *hard-disk*, până la simple erori de programare, cum ar fi accesarea într-un şir a unui element de pe o poziţie mai mare decât lungimea şirului. Când o eroare de acest tip are loc în cadrul unei metode Java, metoda respectivă crează un obiect de tip excepţie pe care îl predă apoi sistemului, sau cu alte cuvinte, maşinii virtuale JVM. Rezultă de aici că excepţiile sunt şi ele nişte clase Java obişnuite. Obiectul de tip excepţie care este predat sistemului conţine informaţii despre excepţia generată, incluzând tipul ei şi starea programului în momentul în care eroarea a apărut. Sistemul este apoi responsabil pentru a găsi secvenţa de cod care tratează excepţia întâlnită. În terminologia Java, crearea unei excepţii şi predarea ei către sistem poartă numele de *aruncare de excepţii* (operaţia `throw`).

După ce excepţia a fost predată sistemului, este sarcina acestuia de a descoperi codul care tratează excepţia respectivă. Sistemul "ştie" unde să caute acest cod, datorită următorului fapt: în Java, apelul unei metode este urmat de păstrarea sa în cadrul unei stive de apel (call stack). Astfel, dacă metoda `main()` a unui program Java apelează o metodă, să spunem `metodaA()`, atunci în stivă se adaugă metoda `main()` urmată de metoda `metodaA()`. Dacă metoda `metodaA()`, apelează o altă metodă `metodaB()`, atunci cea din urmă este adaugată şi ea în stivă. Practic, în acest fel, se poate cunoaşte pentru fiecare metodă în parte, de cine a fost apelată, prin parcurgerea stivei de apel de la vârf spre bază (pentru exemplul nostru, `metodaB()` a fost apelată

Digitized by Google

de `metodaA()`, care la rândul ei a fost apelată de metoda `main()`). Acest mod de a reţine apelul unei metode este cheia soluţiei pe care maşina virtuală o utilizează în tratarea excepţiilor. Maşina virtuală caută în stiva de apel metoda care tratează excepţia, pornind de la metoda în care a apărut eroarea, deci dinspre vârful stivei spre bază. Sistemul consideră căutarea încununată de succes în momentul în care găseşte o metodă din stiva de apel care tratează o excepţie de acelaşi tip cu excepţia aruncată de metoda în care a apărut eroarea. Codul care tratează excepţia respectivă reprezintă, în terminologia Java, codul care *prinde* excepţia (operaţia `catch`). Metoda în care poate apare acest cod, diferă de la un caz la altul. Excepţia poate fi prinsă chiar în metoda în care a fost generată, sau, dimpotrivă, în metodele predecesoare din stiva de apel. Ca exemplu, dacă o excepţie a fost aruncată în `metodaB`, ea poate fi prinsă tot în `metodaB`, sau în `metodaA`, sau în metoda `main`. În majoritatea cazurilor de excepţii este *obligatoriu* ca excepţia aruncată să fie prinsă în cadrul uneia din metodele din stiva de apel. Există însă anumite cazuri, care vor fi prezentate pe parcursul acestui subcapitol, în care se poate întâmpla ca o excepţie să fie aruncată în cadrul unei metode, dar să nu fie prinsă în nici o altă metodă din stiva de apel. În acest caz, programul Java este oprit din execuţie iar excepţia este afişată la terminal.

În concluzie, aruncarea unei excepţii poate fi văzută ca un semnal de alarmă, care indică sistemului că a apărut o problemă pe care trebuie să o trateze pentru ca execuţia programului să poată continua. Semnalul este tras în speranţa că una dintre metodele anterior apelate (cele din stiva de apel) va reacţiona şi va trata problema, iar programul îşi va putea continua astfel execuţia.

6.2 Tipuri de excepţii

În Java, excepţiile sunt obiecte. Când este aruncată o excepţie, este aruncat de fapt un obiect, o instanţă a unei clase de excepţie. Nu orice tip de obiect poate fi însă aruncat ca excepţie. Excepţii pot fi doar obiectele care sunt instanţe ale unor clase derivate din clasa `Throwable`. Clasa `Throwable`, definită în pachetul `java.lang`, reprezintă clasa de bază pentru întreaga familie de clase de excepţii.

Digitized by Google

Figura 6.1: Ierarhia claselor de excepţii ale limbajului Java (pe scurt)

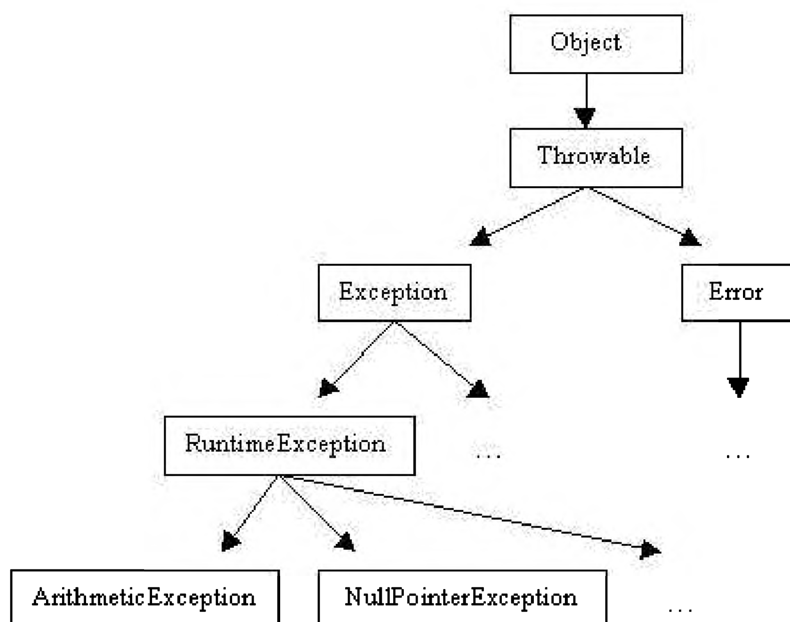


Figura 6.1 ilustrează ierarhia claselor de excepţii, pornind de la clasa de bază `Throwable` şi continuând cu câteva dintre subclasele cele mai importante. După cum se observă, clasa `Throwable` are doi descendenţi direcţi (clase derivate direct din clasa `Throwable`): `Error` şi `Exception`.

Clasa `Error`, împreună cu toate clasele derivate din ea, descrie excepţii grave, numite erori. Erorile reprezintă greşeli de compilare sau probleme mai serioase ale fazei de execuţie a programului, probleme care în general nu pot fi tratate şi necesită întreruperea execuţiei programului (de exemplu, `OutOfMemoryError`). În general, codul scris de programator nu ar trebui să arunce erori, ci doar excepţii. De asemenea, programele Java obişnuite nu ar trebui să prindă erori, deoarece este puţin probabil ca erorile să poate fi în vreun fel tratate.

Clasa `Exception`, împreună cu toate clasele derivate din ea, reprezintă excepţii. Excepţiile sunt aruncate pentru a semnala condiţii anormale, care sunt deseori tratate de aplicaţie, deşi există situaţii în care este posibil ca ex-

Digitized by Google

cepţiile respective să nu fie prinse şi atunci aplicaţia va fi oprită din execuţie. Cele mai multe programe aruncă şi prind instanţe ale unor clase derivate din clasa `Exception`, de aceea `Exception` este clasa cea mai importantă pentru programator din acest punct de vedere. Clasa `Exception` are mulţi descendenţi definiţi în cadrul pachetului `java.lang`. Aceste clase indică diverse tipuri de excepţii care pot apărea în cadrul unui program Java. De exemplu, `NoSuchMethodException` indică faptul că programul a încercat să apeleze o metodă care nu există. Nu toate excepţiile sunt definite în pachetul `java.lang`. Unele dintre ele sunt create pentru a oferi suport pachetelor `net`, `util`, `io` etc. De exemplu, toate excepţiile I/O sunt derivate din `java.io.IOException` şi sunt definite în pachetul `java.io`.

Există o subclasă a clasei `Exception`, care are o însemnătate specială în limbajul Java: `RuntimeException`. Clasa `RuntimeException` este clasa de bază pentru excepţiile care apar în timpul execuţiei unui program (excepţii *runtime*). Pachetele Java definesc multiple clase pentru excepţii *runtime*. Un exemplu des întâlnit este clasa `NullPointerException`, care apare în momentul în care se încearcă accesarea unui membru (atribut sau metodă) al unui obiect, prin intermediul unei referinţe nule.

Costul verificării excepţiilor *runtime* depăşeşte deseori avantajul de a le prinde, motiv pentru care compilatorul Java permite ca excepţiile *runtime* să nu fie prinse sau specificate. De asemenea, excepţiile runtime pot apărea practic în orice secvenţă de cod, motiv pentru care obligativitatea de a le prinde ar fi generat un cod greoi şi obscur, umplut peste măsură cu secvenţe de tratare a excepţiilor. Aceasta este cerinţa pentru ca o excepţie să poată fi aruncată fără a fi prinsă: să fie instanţă a unei clase derivate din `RuntimeException`.

Excepţiile se împart în două mari categorii:

- tratate (checked);
- netratate (unchecked).

Excepţiile *tratate* sunt denumite astfel pentru că atât compilatorul Java cât şi maşina virtuală JVM verifică dacă excepţiile aruncate în cadrul unei metode sunt fie tratate în cadrul acelei metode, fie specificate ca fiind aruncate mai departe de metoda respectivă, urmând a fi tratate în alte metode.

O clasă de excepţii este *tratată* sau *netratată*, în funcţie de poziţia ei în ierarhia claselor de excepţii (cu clasa de bază `Throwable`). Potrivit acesteia, se consideră excepţie *tratată*, orice clasă derivată din clasa `Exception`, inclusiv clasa `Exception`, mai puţin clasa `RuntimeException` şi subclasele ei. Toate celelalte clase de excepţii (clasa `Throwable`, clasa `Error` şi subclasele

Digitized by Google

ei), împreună cu clasa `RuntimeException` şi subclasele ei, reprezintă excepţii *netratate*.

Diferenţa conceptuală dintre excepţiile tratate şi cele netratate este că primele semnalează situaţii anormale de care programatorul este obligat să ţină seama. Cu alte cuvinte, aruncarea excepţiilor tratate în cadrul unei metode, obligă programatorul să le prindă şi să le trateze mai devreme sau mai târziu în cadrul aplicaţiei sale. Compilatorul este cel care îl obligă pe programator să trateze excepţiile respective. În cazul excepţiilor netratate, programatorul poate decide dacă prinde sau trece cu vederea excepţia. De altfel, este posibil ca programatorul nici să nu ştie că excepţia netratată poate fi aruncată în cadrul metodei. Ca o consecinţă, este de la sine înţeles că nici compilatorul nu obligă programatorul să trateze excepţia respectivă.

Utilizarea excepţiilor nu este nicidecum limitată doar la cele definite de platforma Java. Pe lângă aceste clase, programatorul poate defini propriile clase de excepţii. Acest lucru este foarte important, pentru că deseori un programator simte nevoia să creeze propriile sale excepţii, pentru a înfăţişa o situaţie specială de eroare, pe care programul său este capabil să o producă, dar care nu a fost prevăzută la crearea ierarhiei de excepţii. Această situaţie este perfect normală, deoarece creatorii limbajului Java nici nu ar fi putut să anticipeze toate situaţiile de eroare ce pot apare în aplicaţii existente sau viitoare. Crearea propriilor excepţii este chiar încurajată, deoarece regula generală este ca programul să fie cât mai precis. De aceea, programatorii nu trebuie să utilizeze excepţii ambigue, de tipul `Exception`, ci subclase ale acestora, pentru a nuanţa cât mai bine excepţia apărută. De exemplu, dacă în cadrul programului nostru a fost citit un nume de utilizator invalid, atunci este de preferat să definim, să aruncăm şi să prindem o excepţie de genul `InvalidUserNameException`. În continuare, vor fi prezentate cele mai importante amănunte legate de crearea propriilor clase de excepţii.

6.3 Definirea propriilor excepţii de către programator

Java oferă două posibilităţi importante de a utiliza excepţii într–un program:

- utilizarea unor excepţii predefinite sau definite de alţi programatori (de exemplu, clasele de excepţii cuprinse în platforma Java) ;

- utilizarea unor excepţii proprii, definite de programator.

Digitized by Google

Pachetele oferite de platforma Java oferă numeroase clase de excepţii. Toate aceste clase sunt derivate direct sau indirect din clasa `Throwable` şi permit programatorului să diferenţieze diversele tipuri de excepţii care pot să apară de-a lungul execuţiei unui program.

Platforma Java permite şi crearea propriilor clase de excepţii pentru a înfăţişa situaţii problematice specifice care pot să apară în cadrul programelor. Deşi necesitatea de a avea clase proprii de excepţii poate să pară exagerată, totuşi aceasta devine evidentă în cazul în care clasele scrise de programator vor fi reutilizate de către alţi programatori. În astfel de situaţii, programatorul trebuie să îşi creeze propria structură de clase de excepţii pentru a permite utilizatorilor claselor sale să diferenţieze o eroare apărută în clasele sale, de erori apărute în clase scrise de alţi programatori. Un alt motiv pentru care programatorii apelează la definirea de excepţii proprii este nevoia de a avea o excepţie care nu există în ierarhia de clase de excepţii oferită de platforma Java, şi care descrie o situaţie de excepţie particulară aplicaţiei respective.

Ca şi clasele de excepţii predefinite de platforma Java, clasele proprii de excepţii trebuie să extindă (direct sau indirect) clasa `Throwable`. O primă tentaţie ar fi aceea de a extinde direct clasele proprii de excepţie din clasa `Throwable`. La o privire mai atentă, se poate observa că `Throwable` are două subclase, `Error` şi `Exception`, care împart în două erorile posibile dintr-un program Java. Marea majoritate a programelor Java utilizează excepţii derivate din `Exception`, deoarece excepţiile de tip `Error` sunt rezervate pentru probleme mult mai complexe care au loc în cadrul sistemului (maşinii virtuale JVM). În definirea noilor excepţiilor, se consideră o bună practică adăugarea la numele clasei a sufixului `Exception` pentru toate clasele derivate (direct sau indirect) din `Exception` (de exemplu, `TeaTooHotException`). Evident, analog se procedează şi în cazul (mult mai rar) al claselor derivate (direct sau indirect) din `Error`.

Deoarece clasele proprii de excepţii sunt clase Java obişnuite, ele pot îngloba informaţii despre situaţia anormală care a condus la aruncarea lor. Astfel, se pot transmite informaţii utile către partea de program care tratează respectiva situaţie de eroare.

În general, clasele proprii de excepţii sunt implementate astfel:

Listing 6.1: Clasa `TeaTooHotException`

```
public class TeaTooHotException extends Exception
{
    public TeaTooHotException()
    {
        super();
    }
```


Digitized by Google

```

    public TeaTooHotException(String s)
    {
        super(s);
    }
}
```

Clasa `TeaTooHotException` nu aduce practic nimic nou clasei `Exception`, deoarece defineşte doar doi constructori care apelează constructorii din clasa de bază. Totuşi, *tipul* excepţiei în sine este foarte important: el ne ajută să discernem în momentul în care prindem excepţia despre ce fel de eroare este vorba.

Folosind definirea anterioară, utilizatorul clasei de excepţii va putea arunca o excepţie în două moduri:

```
new TeaTooHotException();

new TeaTooHotException("Ceaiul este foarte fierbinte");
```

Este uşor de observat că în cel de al doilea caz situaţia de eroare transmite şi o informaţie, sub forma mesajului `"Ceaiul este foarte fierbinte"`, care poate fi folosit de codul care tratează excepţia apărută pentru a afişa un mesaj informativ.

Informaţiile transmise de excepţii nu se limitează doar la mesaje de tip `String`. Dacă mesajul `String` nu este suficient pentru a îngloba întreaga informaţie dorită, se pot adăuga date noi clasei de excepţii, sub forma atributelor şi metodelor.

Listing 6.2 prezintă o variantă modificată a clasei `TeaTooHotException` care înglobează în informaţiile transmise şi temperatura ceaiului (engl. tea) din exemplul nostru.

Listing 6.2: Clasa `TeaTooHotException` (versiune modificată)

```
public class TeaTooHotException extends Exception
{
  int temperature;

  public TeaTooHotException()
  {
    super();
  }

  public TeaTooHotException(String s)
  {
    super(s);
  }
```


Digitized by Google

```java

    public TeaTooHotException(int temperature)
    {
        this.temperature = temperature;
    }

    public int getTemperature()
    {
        return temperature;
    }
}
```

Constructorul cu argument de tip `int` ne permite să creăm o excepţie care înglobează ca informaţie temperatura ceaiului prea fierbinte (engl. too hot) pentru a fi consumat. Iată un exemplu de utilizare a acestui constructor:

```java
new TeaTooHotException(70);
```

În momentul în care excepţia este prinsă şi tratată, undeva în cadrul aplicaţiei, se vor putea obţine informaţii suplimentare referitoare la temperatura care a provocat excepţia `TeaTooHotException`, utilizând metoda `getTemperature()`.

În cele mai multe cazuri o asemenea abordare nu este necesară şi singura informaţie (esenţială) legată de excepţie este tipul obiectului creat (cu alte cuvinte, numele clasei), fără alte informaţii adiacente stocate în cadrul obiectului respectiv.

6.4 Prinderea şi tratarea excepţiilor

Când o metodă aruncă excepţii, ea presupune că acestea sunt prinse şi tratate undeva în cadrul aplicaţiei. Pentru prinderea şi tratarea lor, Java pune la dispoziţia programatorilor trei cuvinte cheie, care vor fi descrise în continuare:

- `try`
- `catch`
- `finally`

6.4.1 Blocul `try`

Primul pas în prinderea şi tratarea excepţiilor este acela de a "îngrădi" instrucţiunile care pot genera excepţii într-un bloc de instrucţiuni, denumit bloc `try`. Blocul `try` cuprinde instrucţiunile care pot arunca excepţii. În general, un bloc `try` arată astfel:

Digitized by Google

```
try
{
    //instructiuni care pot genera exceptii
    instructiuni;
}
```

Blocul `try` ţine sub observaţie toate instrucţiunile pe care le conţine, în aşteptarea unei excepţii aruncate de una dintre respectivele instrucţiuni. El poate fi văzut ca un bloc de instrucţiuni care sunt "încercate" (try = a încerca), urmând ca, în cazul în care ele nu se pot executa cu succes, eroarea să fie tratată. Evident că, dacă instrucţiunile se execută fără a genera probleme, nu mai poate fi vorba de tratarea excepţiilor şi aplicaţia se va executa normal, fără a ţine cont de partea de tratare a excepţiilor. Acesta reprezintă un mare avantaj oferit de modul de gestionare a erorilor în limbajul Java. Astfel, programatorului îi este permis să se concentreze asupra problemei pe care încearcă să o rezolve, fără a-şi pune problema eventualelor erori care pot fi generate de codul lui. Aceste erori sunt tratate ulterior. Din acest motiv, codul devine mult mai uşor de scris şi de citit, deoarece scopul pentru care codul a fost scris nu se mai confundă cu verificarea eventualelor erori.

Unui bloc `try` i se asociază obligatoriu unul sau mai multe blocuri `catch` şi, opţional, un bloc `finally`.

6.4.2 Blocul `catch`

Blocul `catch` reprezintă locul în care sunt prinse şi tratate excepţiile. Blocurile `catch` nu pot exista independent, ele apar doar în asociere cu un bloc `try`:

```
try
{
    ...
}
catch (...)
{
    ...
}
catch (...)
{
    ...
}
...
```

Între blocul `try` şi primul bloc `catch` nu pot exista instrucţiuni intermediare. Forma generală a unui bloc `catch` este următoarea:

```
catch(ClasaDeExceptii numeVariabila)
```


Digitized by Google

```
{
    instructiuni;
}
```

După cum se poate observa, instrucţiunea `catch` necesită un singur parametru formal, notat `numeVariabila`. Modul de definire a instrucţiunii `catch` este foarte asemănător cu cel de definire a argumentelor unei metode Java, unde lista argumentelor unei metode era compusă din perechi de tipul *numeClasa* şi *numeVariabila*, separate prin simbolul ",". În cazul excepţiilor, `ClasaDeExceptii` reprezintă tipul de excepţie care poate fi tratat, specificat prin numele clasei de excepţie, derivată din `Throwable`. Parametrul formal, `numeVariabila`, reprezintă numele prin care poate fi referită excepţia tratată de blocul `catch`. Excepţia poate fi referită prin acest nume doar în cadrul blocului `catch`, pentru că durata de viaţă a variabilei `numeVariabila` se rezumă doar la blocul `catch`. Deseori variabila `numeVariabila` nu este utilizată, pentru că tipul clasei de excepţie oferă îndeajuns de multă informaţie programatorului. Totuşi, numele variabilei trebuie întotdeauna specificat, chiar dacă nu este utilizat niciodată în cadrul blocului `catch`. Pentru un exemplu de utilizare a blocurilor `try-catch`, să presupunem că încercăm accesarea unui element într-un şir:

```
int[] s = {4, 1, 8, 7};
try
{
    System.out.println(s[6]);
}
catch(ArrayIndexOutOfBoundsException exc)
{
    exc.printStackTrace();
}
```

Încercarea de a accesa un element care nu există în şirul `s` (acesta are 4 elemente, deci nu poate fi vorba de un element pe poziţia 6) va arunca o excepţie, `ArrayIndexOutOfBoundsException`. Aceasta este una dintre multiplele excepţii predefinite de limbajul Java, derivată indirect din clasa `Throwable` şi aflată în pachetul `java.lang`. Excepţia astfel aruncată este prinsă de blocul `catch` asociat, care specifică faptul că tratează excepţii `ArrayIndexOutOfBoundsException`. Faptul că excepţia aruncată a fost prinsă de acest bloc `catch`, presupune că tot acest bloc o va şi trata. Prin urmare, se execută instrucţiunile cuprinse în cadrul blocului `catch`, care sunt create pentru a trata cazul de eroare apărut. Excepţia apărută este referită în cadrul blocului `catch` prin intermediul numelui `exc`. Variabila `exc` este o instanţă a clasei `ArrayIndexOutOfBoundsException`, care este un obiect Java cu metode şi atribute. Deoarece toate clasele de excepţii sunt derivate din clasa

Digitized by Google

`Throwable`, ele moştenesc câteva metode utile pentru aflarea de informaţii despre situaţia de eroare. Printre aceste metode se află `printStackTrace()`, `getMessage()`, `toString()`, fiecare dintre ale afişând informaţii care diferă în cantitate şi calitate de la un caz la altul. Pentru exemplul nostru, instrucţiunea `exc.printStackTrace()` va afişa următorul mesaj (`Exc.java` este numele fişierului care conţine exemplul):

```
java.lang.ArrayIndexOutOfBoundsException
        at Exc.main(Exc.java:8)
```

Instrucţiunile din blocul `catch` se execută doar dacă instrucţiunile din blocul `try` generează excepţii şi aceste excepţii sunt instanţe ale clasei predefinite `ArrayIndexOutOfBoundsException`. Este de la sine înţeles că dacă s-ar fi încercat accesarea unui element de pe o poziţie mai mică decât lungimea şirului (de exemplu `s[2]`), nu ar mai fi fost o situaţie de eroare şi, implicit, excepţia `ArrayIndexOutOfBoundsException` nu ar mai fi fost aruncată, iar instrucţiunea din cadrul blocului `try` s-ar fi executat corect, afişând elementul de pe poziţia `2` din şir (adică, `8`).

Pentru ca un bloc `catch` să prindă şi să trateze o excepţie, nu este necesar ca tipul de excepţie tratat de blocul `catch` şi tipul de excepţie aruncat de o metodă din blocul `try` să fie identice. Un bloc `catch`, care tratează un tip de excepţii, va prinde şi trata şi clasele de excepţii derivate din excepţia respectivă. Cu alte cuvinte, în exemplul de faţă, nu este necesar să prindem excepţia creată folosind clasa `ArrayIndexOutOfBoundsException`. Deoarece această clasă de excepţii este derivată din `Exception`, codul ar putea arăta şi aşa:

```
int[] s = {4, 1, 8, 7};
try
{
    System.out.println(s[6]);
}
catch(Exception exc)
{
    exc.printStackTrace();
}
```

În această situaţie, blocul `catch` prinde toate excepţiile `Exception`, dar şi clase de excepţii derivate din clasa `Exception`, cum este şi cazul clasei de excepţii `ArrayIndexOutOfBoundsException`. Practic, în acest mod, se prind mai multe excepţii cu acelaşi bloc `catch`, deoarece acest bloc prinde toate excepţiile de tip `Exception`, plus clasele de excepţii derivate din aceasta. Dacă se foloseşte această abordare, de a avea un bloc `catch` pentru mai multe excepţii generate, atunci codul de tratare a situaţiei de eroare devine

Digitized by Google

prea general, pentru că excepţia tratată de blocul `catch` nu este specializată, cu alte cuvinte este o clasă de bază pentru alte clase de excepţii. Din această cauză, se preferă utilizarea blocurilor `catch` specializate, şi nu a celor generale, pentru că ele pot oferi mai multe detalii despre situaţia de eroare. Totuşi, cazul `Exception` este şi el important, pentru că astfel se pot prinde şi trata toate tipurile de excepţii aruncate într-un mod unitar (`Exception` este clasa de bază a tuturor excepţiilor de programare).

Un alt aspect important care trebuie înţeles în legătură cu tratarea excepţiilor este că Java presupune că problema care a generat excepţia este atât de critică, încât nu se poate realiza întoarcerea la instrucţiunea care a generat-o, cu alte cuvinte nu se poate reface situaţia dinaintea apariţiei erorii. Acest aspect este foarte important pentru cei care cred că tratarea unei excepţii presupune realizarea unor operaţii care să îndrepte situaţia, iar apoi metoda care a generat-o să fie executată din nou, în speranţa că a doua tentativă va fi încununată de succes. Java *nu* realizează acest lucru. După ce excepţia a fost tratată, se execută în continuare următoarea linie de cod aflată după secvenţa `catch`. Totuşi, se poate folosi un mic truc, care să ofere această posibilitate, şi anume, instrucţiunea care generează eroarea să fie plasată în cadrul unui ciclu (`for`, `while`, `do`).

Este foarte posibil ca într–un bloc `try` să existe instrucţiuni care pot genera excepţii de mai multe tipuri. Aceste excepţii pot fi tratate separat, asociind blocului `try` mai multe blocuri `catch`. De exemplu, putem avea o metodă `drinkTea()`, care aruncă două excepţii, `TeaTooHotException` şi `TeaTooColdException`, ambele clase fiind derivate din `Exception`. În consecinţă, am putea avea următorul cod:

```
//temperatura ceaiului
int temperature = 25;

try
{
    drinkTea(temperature);
}
catch(TeaTooHotException hotExc)
{
    hotExc.printStackTrace();
}
catch(TeaTooColdException coldExc)
{
    coldExc.printStackTrace();
}
```

Utilizarea mai multor blocuri `catch` asociate unui bloc `try` trebuie să ţină cont de anumite reguli, care vor fi prezentate în continuare.

Digitized by Google

Blocurile `catch` sunt examinate în ordinea în care ele apar în fişierul sursă. În timpul procesului de examinare, primul bloc care este în măsură să trateze eroarea, o va face, iar celalte blocuri nu vor mai fi luate în considerare, deşi ele ar putea, la rândul lor, să trateze respectiva excepţie. De aceea, ordinea blocurilor `catch` contează. Totodată, un bloc care tratează o excepţie generală nu se poate afla înaintea unui bloc care tratează o excepţie specializată, derivată din cea generală, deoarece secvenţa de tratare specializată nu ajunge să fie executată niciodată. Pentru exemplul nostru, putem considera excepţia generală `Exception`, iar excepţia specializată ca fiind `TeaTooHotException`, care este derivată din excepţia generală. Ordinea blocurilor `catch` este obligatoriu următoarea:

```
//temperatura ceaiului
int temperature = 25;

try
{
    drinkTea(temperature);
}
catch(TeaTooHotException hotExc)
{
    hotExc.printStackTrace();
}
catch(Exception exc)
{
    exc.printStackTrace();
}
```

Ordinea este logică, pentru că, dacă ar fi fost posibilă inversarea ordinii celor două excepţii, atunci `Exception` ar fi fost tot timpul excepţia prinsă, în timp ce `TeaTooHotException` ar fi fost imposibil de prins, deci codul de tratare a acestei excepţii nu ar fi fost executat niciodată. Aşadar, regula generală este că blocurile `catch` care tratează subclase de excepţii trebuie să precedeze blocurile `catch` ce tratează superclase.

6.4.3 Blocul `finally`

Din momentul în care maşina virtuală JVM începe să execute un bloc `try`, ea poate avea mai multe posibilităţi de a ieşi din acel bloc. Ar putea întâlni în cadrul blocului instrucţiuni de tipul `break`, `continue`, `return`, care să cauzeze ieşirea prematură din bloc. Pe de altă parte, diferitele excepţii aruncate de codul din blocul `try` ar putea avea drept rezultat o altă întrerupere prematură a executării codului din acel bloc.

Există multe situaţii în care este de o importanţă vitală să se execute anumite

Digitized by Google

instrucţiuni la ieşirea din blocul `try`, indiferent de modul în care această ieşire se realizează. De exemplu, dacă într-un bloc `try` este deschis un fişier pentru a scrie date în el, este de dorit o secvenţă de cod care să îl închidă, indiferent de cum se termină execuţia blocului de scriere. Existenţa unei astfel de secvenţe de cod este necesară şi în situaţia în care se doreşte eliberarea anumitor resurse utilizate în blocul `try`, cum ar fi fluxurile de date etc. Pentru a realiza acest deziderat, mediul de programare Java oferă noţiunea `finally`.

Clauza `finally` poate fi utilizată doar în asociere cu un bloc `try`. Ea constă, ca şi blocul `try`, dintr-un bloc de instrucţiuni care permite ca o secvenţă de cod să fie executată, indiferent dacă blocul `try` aruncă sau nu excepţii. Prin clauza `finally`, instrucţiunea `try-catch` devine completă, având următoarea sintaxă:

```
try
{
    //instructiuni care pot arunca exceptii
    ...
}
catch(E1 e1)
{
    //instructiuni care trateaza exceptii de tipul E1
    //si exceptii derivate din clasa de exceptii E1
    ...
}
catch(E2 e2)
{
    //instructiuni care trateaza exceptii de tipul E2
    //si exceptii derivate din clasa de exceptii E2
    ...
} //numarul de blocuri catch este variabil
finally
{
    //instructiuni care se executa indiferent de ce
    //se intampla in blocul try
    ...
}
```

Pentru o mai bună înţelegere a blocului `finally` să considerăm programul din **Listing 6.3**, în care avem o aplicaţie care deschide (sau crează dacă nu există) un fişier stocat pe hard-disk şi scrie în el un şir de numere întregi.

Listing 6.3: Program de test pentru clauza `finally`

```
//Nume fisier: TestFinally.java

import java.io.*;

public class TestFinally
```


Digitized by Google

```
{
  public static void main(String[] args)
  {
    PrintWriter pw = null;
    int[] s = {3, 7, 4, 6};
    String msg = "";

    try
    {
      msg = "Se incearca deschiderea fisierului...";
      System.out.println(msg);
      pw = new PrintWriter(new FileWriter(
      "fisier.txt"));

      for (int i = 0; i < s.length; i++)
      {
        pw.println(s[i]);
      }
    }
    catch(IOException ioe)
    {
      msg = "Exceptie I/O prinsa: " + ioe.getMessage();
      System.out.println(msg);
    }
    finally
    {
      if (pw != null)
      {
        pw.close();
        System.out.println("Fisierul a fost inchis");
      }
      else
      {
        System.out.println("Fisierul nu a fost " +
        "deschis");
      }
    }
  }
}
```

Programul afişează următoarele mesaje:

```
Se incearca deschiderea fisierului...
Fisierul a fost inchis
```

Totodată, programul crează un fişier numit `fisier.txt`, în acelaşi director cu fişierul sursă al aplicaţiei noastre, cu următorul conţinut:

```
3
```


Digitized by Google

```
7
4
6
```

La o analiză atentă se observă că instrucţiunile din blocul `try` nu au generat nici o excepţie (nu se afişează mesajul `"Exceptie I/O prinsa:  "`), deci totul s-a executat fără eroare, iar ieşirea din blocul `try` nu a fost bruscă. Totuşi, instrucţiunile din blocul `finally` s-au executat pentru că a fost afişat mesajul `"Fisierul a fost inchis"`.

Să încercăm acum să înlocuim numele fişierului, `fisier.txt`, cu o cale invalidă, cum ar fi `C:\test\fisier.txt`, cu condiţia ca directorul `test` să NU existe pe partiţia `C:` a hard-disk-ului dumneavoastră. Rulând programul astfel modificat, se obţin următoarele rezultate:

```
Se incearca deschiderea fisierului...
Exceptie I/O prinsa: c:\test\fisier.txt (The system
cannot find the path specified)
Fisierul nu a fost deschis
```

În această situaţie, constructorul clasei `FileWriter` aruncă o excepţie de tipul `IOException`, pentru că nu poate găsi directorul `test`, şi în consecinţă nu poate crea fişierul `fisier.txt`. Excepţia aruncată este tratată de blocul `catch` existent, care afisează mesajul `"Exceptie I/O..."`, iar apoi, se execută din nou instrucţiunile din blocul `finally`, afişându-se mesajul `"Fisierul nu a fost deschis"`.

După cum se poate observa, blocul `finally` a fost executat indiferent dacă situaţia a fost de eroare (al doilea caz) sau nu (primul caz). Deoarece în blocul `finally` au fost eliberate resursele alocate în blocul `try` (în cazul nostru, fluxul de scriere `PrintWriter`), putem considera blocul `finally` un mecanism elegant de a "face curăţenie" după execuţia codului din blocul `try`.

Situaţia de eroare prezentată anterior nu este singulară. `IOException` pot fi generată şi dacă directorul respectiv există (deci calea este validă), dar aplicaţia nu are drept de scriere în acel director, sau dacă nu mai există suficient spaţiu pe disc pentru a scrie date în fişierul respectiv.

Cu siguranţă că programatorii cu experienţă în alte limbaje de programare (în special C++, unde această noţiune nu există) se vor întreba dacă existenţa clauzei `finally` este justificată. Nevoia de a avea clauza `finally` devine evidentă dacă analizăm mai atent programul anterior, în care, fără `finally`, fluxul de scriere ar fi putut fi eliberat doar dacă am fi plasat codul în fiecare bloc `catch` separat, ceea ce ar fi dus la o duplicare inutilă a codului.

Digitized by Google

6.5 Aruncarea excepţiilor

Înainte ca o excepţie să fie prinsă, trebuie să existe, undeva în cadrul claselor utilizate, o secvenţă de cod care să arunce excepţia respectivă. Excepţiile pot fi aruncate atât de codul scris de dumneavoastră, cât şi codul din clase scrise de alţi programatori (cum ar fi, excepţiile aruncate de clasele predefinite în platforma Java), sau chiar de maşina virtuală JVM. Indiferent de cine aruncă excepţia, ea este întotdeauna aruncată folosind cuvântul cheie `throw`.

6.5.1 Instrucţiunea `throw`

Metodele Java aruncă excepţii folosind instrucţiunea `throw`. Instrucţiunea `throw` este urmată de un singur argument, care trebuie să fie o instanţă a unei clase derivate din clasa `Throwable`, şi are următoarea formă:

```
throw instantaClasaDeExceptii;
```

Iată câteva exemple de utilizare:

- ```
  throw new Exception();
  ```

- ```
  Exception e = new Exception();
  throw e;
  ```

- ```
  //temperatura initiala a ceaiului
  int temperature = 30;

  //calcule intermediare
  ....

  //consideram ca temperatura la care ceaiul
  //nu este inca fierbinte, este de 40 de grade
  if (temperature > 40)
  {
      //ceaiul este foarte fierbinte si nu poate fi baut,
      //deci suntem intr-o situatie de eroare,
      //motiv pentru care aruncam o exceptie
      throw new TeaTooHotException();
  }
  ```


Digitized by Google
  ```

Încercarea de a arunca drept excepţie un obiect care nu este instanţă a unei clase de excepţie, este semnalată drept eroare de către compilator.

În momentul aruncării unei excepţii se realizează mai multe lucruri: mai întâi, se crează obiectul de tip excepţie, în acelaşi mod în care se crează orice obiect Java, prin folosirea cuvântului cheie `new`. Apoi, se întrerupe execuţia normală a instrucţiunilor, iar mecanismul de tratare a excepţiilor, aflat în JVM, preia sarcina de a căuta secvenţa de cod care tratează excepţia respectivă, pentru a putea continua execuţia programului.

Crearea obiectului de tip excepţie constă de fapt în apelul unui constructor al clasei de excepţie. Aceasta poate avea mai multe tipuri de constructori, mai ales dacă respectiva clasă de excepţie este chiar o excepţie proprie, scrisă de dumneavoastră (paragraful 6.3). În mod obişnuit sunt două tipuri de constructori care pot fi apelaţi: constructorul implicit, care nu are parametri, şi constructorul cu un mesaj de tip `String`. De exemplu:

```
throw new TeaTooHotException();

throw new TeaTooHotException("Ceaiul este foarte " +
                             "fierbinte.");
```

De obicei, se aruncă câte o clasă diferită de excepţie pentru fiecare tip de eroare întâlnit. Având în vedere tipurile de constructori ai clasei de excepţie, informaţia despre eroare este reprezentată atât în interiorul obiectului de excepţie (cu ajutorul unui mesaj `String` etc.), cât şi de tipul obiectului de excepţie (cu alte cuvinte, de numele clasei de excepţie).

Uneori este necesar ca o anumită excepţie să fie rearuncată, adică să fie aruncată exact în locul în care a fost prinsă. Iată un exemplu de rearuncare a unei excepţii:

```
try
{
    ...
}
catch(Exception e)
{
    throw e;
}
```

Prinderea şi rearuncarea aceleiaşi excepţii (ca în exemplul anterior) nu are nici un sens, deoarece este ca şi cum am ignora excepţia respectivă. Situaţia este diferită însă când tipul excepţiei prinse nu este neapărat acelaşi cu tipul excepţiei rearuncate. Asemenea situaţii apar frecvent în aplicaţiile ceva mai complexe în care o excepţie de un anumit gen este "tradusă" într-o excepţie de alt gen, dar cu conţinut echivalent (de exemplu, într-o aplicaţie client-server, o excepţie prin care serverul anunţă că nu a putut procesa o anumită cerere a clientului,

Digitized by Google

este tradusă de programul client într-o altă excepţie care să conţină un mesaj inteligibil pentru utilizator). În exemplul următor, `MyNewException` este o clasă excepţie proprie aruncată în cazul prinderii unei excepţii de intrare-ieşire:

```
try
{
    ...
}
catch(IOException e)
{
    throw new MyNewException();
}
```

6.5.2 Clauza `throws`

Orice metodă Java trebuie fie să prindă, fie să specifice clar toate excepţiile pe care le aruncă mai departe metodelor care o apelează. Este vorba, desigur, de excepţiile tratate (adică, cele care nu sunt derivate din `RuntimeException`, vezi paragraful 6.2) care pot fi aruncate de codul din interiorul metodei respective.

Am prezentat până acum doar cazul în care excepţiile sunt prinse şi tratate în cadrul aceleaşi metode, nefiind transmise mai departe. Această secţiune va prezenta cea de a doua posibilitate: specificarea excepţiilor care nu sunt tratate în cadrul metodei respective, fiind aruncate mai departe, pentru a fi prinse şi tratate de alte metode (cu alte cuvinte, vom învăţa cum se aruncă pisica moartă în ograda vecinului).

Atunci când decidem ca o metodă să nu prindă o anumită excepţie care poate fi aruncată de codul din interiorul ei, trebuie să precizăm faptul că metoda poate arunca la rândul ei excepţia respectivă (deci vecinii trebuie avertizaţi în legătură cu posibilitatea de a se trezi cu o pisică decedată în ogradă). Această cerinţă este perfect explicabilă, pentru că specificarea excepţiilor aruncate mai departe de metodă ţine de interfaţa metodei, cu alte cuvinte cei care utilizează o metodă trebuie să cunoască precis ce excepţii aruncă chiar şi în cazul în care nu au la dispoziţie codul sursă al metodei, pentru a putea elabora o strategie de tratare a respectivelor excepţii.

Specificarea excepţiilor aruncate de o metodă se realizează prin intermediul clauzei `throws`, utilizată în antetul metodei respective:

```
tipReturnat numeMetoda(listaArgumente) throws Exception1, ...
{
    ...
}
```


Digitized by Google

Clauza `throws` se foloseşte în situaţia în care programatorul decide că nu este potrivit ca metoda în cauză să prindă şi să trateze o excepţie aruncată de o altă metodă pe care o apelează, sau dacă metoda aruncă propriile sale excepţii.

Lista excepţiilor specificate în clauza `throws` cuprinde excepţiile aruncate direct, folosind instrucţiunea `throw`, precum şi excepţiile (netratate) aruncate de alte metode apelate. Evident, ambele tipuri de excepţii specificate în clauza `throws`, nu trebuie să fie tratate în cadrul metodei. Dacă acest lucru se întâmplă, atunci excepţiile nu mai pot fi precizate în clauza `throws`.

Iată un exemplu de specificare a excepţiilor care nu sunt prinse şi tratate în cadrul unei metode:

```
public void drinkTea(int temp) throws TeaTooHotException
{
    //daca temperatura e prea mare
    if (temp > 40)
    {
        throw new TeaTooHotException();
    }
}
```

Exemplul de faţă specifică faptul că metoda `drinkTea()` aruncă o excepţie proprie, `TeaTooHotException`, dacă temperatura ceaiului este mai mare de 40 de grade. Datorită acestui lucru, apelul metodei `drinkTea()` întro altă metodă se poate realiza doar dacă excepţia `TeaTooHotException` este prinsă şi tratată în cadrul acelei metode, sau dacă şi acea metodă alege să arunce excepţia mai departe, specificând-o în clauza `throws`:

- ```
  public void serveCustomer()
  {
      int temperature = 45;

      try
      {
          drinkTea(temperature);
      }
      catch(TeaTooHotException exc)
      {
          ;
      }
  }
  ```

- ```
  public void serveCustomer() throws TeaTooHotException
  {
  ```


Digitized by Google

```
    int temperature = 45;

    drinkTea(temperature);
}
```

Este evident că în cel de-al doilea caz, excepţia va trebui prinsă şi tratată la un moment dat, în cadrul unei metode. Deoarece este o excepţie tratată, ea nu poate fi aruncată mai departe de fiecare metodă, fără să existe în final o metodă care să o prindă şi să o trateze.

Specificarea excepţiilor aruncate de o metodă nu se poate evita, dacă metoda respectivă generează excepţii pe care nu le tratează, deoarece compilatorul detectează această "scăpare" şi o raportează drept eroare de compilare, propunând două soluţii: tratarea lor în cadrul metodei sau specificarea lor în clauza `throws`.

Totuşi există o posibilitate de a "păcăli" compilatorul, atunci când se specifică faptul că o metodă aruncă o anumită excepţie, deşi acest lucru nu se întâmplă. În această situaţie, compilatorul obligă apelurile metodei noastre să trateze excepţia sau să o specifice în clauza `throws` a metodei apelante, chiar dacă metoda noastră nu aruncă propriu-zis excepţia respectivă, ci doar pretinde că o aruncă. Acest lucru este util în momentul în care ştim că în viitor metoda noastră va fi modificată astfel încât să arunce respectiva excepţie, deoarece nu va mai fi necesară modificarea codului care apelează metoda, codul respectiv tratând excepţia în avans, înainte ca ea să fie aruncată propriu-zis de metodă.

O ultimă observaţie: mecansimul de aruncare mai departe a excepţiilor oferă o modalitate deosebit de elegantă de a trata problemele care apar în decursul execuţiei la nivelul adecvat. De exemplu, este foarte posibil ca o metodă care citeşte un flux de octeţi să întâlnească diverse probleme, cum ar fi: fluxul nu poate fi deschis, transmiterea de date s-a întrerupt prematur etc. Totuşi, metoda respectivă este de un nivel prea jos pentru a putea lua decizii în legătură cu acţiunile care trebuie întreprinse în cazul unei erori (care pot chiar diferi de la o situaţie la alta). Aruncând o excepţie, metoda noastră are posibilitatea de a anunţa metodele de nivel mai înalt care au apelat-o de faptul că secvenţa de execuţie nu a decurs normal, lăsând în seama acestora modul în care se tratează eroarea. Programatorii cu ceva experienţă în C sau Pascal vor recunoaşte probabil superioritatea acestei metode, în faţa celei în care metoda care citea dintr-un flux de date seta o variabilă sau returna o anumită valoare care descria modul în care a decurs citirea din flux (de exemplu, returna 0 dacă citirea a decurs normal, 1 dacă nu s-a putut deschide fişierul, 2 dacă citirea s-a întrerupt etc.).

Digitized by Google

6.6 Sugestii pentru utilizarea eficientă a excepţiilor

Problemele cele mai mari cu care se confruntă programatorii atunci când utilizează excepţiile Java sunt *când* şi *ce* excepţii să arunce.

Pentru prima problemă, răspunsul sumar pe care un programator ar putea să-l ofere, ar arăta astfel:

> Dacă o metodă întâlneşte o situaţie anormală pe care nu o poate trata, atunci metoda respectivă ar trebui să arunce o excepţie.

Din nefericire, acest răspuns este cât se poate de ambiguu, pentru că ne conduce către o altă întrebare: ce înseamnă o "situaţie anormală"? Aceasta poate fi considerată întrebarea cheie a tratării excepţiilor.

A decide dacă o anumită situaţie este anormală sau nu, rămâne însă o problemă subiectivă. Decizia nu este întotdeauna evidentă şi de aceea depinde în mare măsură şi de programator. Totuşi, situaţia nu este chiar atât de gravă pe cât aţi fi tentaţi să credeţi. Pentru a veni în ajutorul programatorilor, creatorii limbajului Java au considerat că o bună regulă de utilizare a excepţiilor este aceea de a "evita utilizarea excepţiilor pentru a indica situaţii previzibile în funcţionarea standard a unei metode".

În consecinţă, o situaţie anormală este un eveniment care nu poate fi prevăzut în cazul unei funcţionări normale a metodei. Pentru o mai bună înţelegere a acestui concept, vom considera câteva exemple utile, în care se decide dacă este utilă sau nu utilizarea excepţiilor. Ca exemple am ales câteva situaţii întâlnite chiar de dezvoltatorii limbajului Java la crearea ierarhiei de clase oferită de platforma Java, pentru a vedea cum au fost acestea tratate (dacă au fost considerate situaţii anormale şi, implicit, s-a recurs la utilizarea excepţiilor, sau dacă au fost considerate situaţii normale, previzibile).

Să considerăm mai întâi cazul metodei `read()` a clasei `FileInputStream`, disponibilă în pachetul `java.io`. Această metodă returnează un `byte` citit dintr-un flux de intrare specificat. Situaţia specială cu care s-au confruntat dezvoltatorii limbajului Java a fost următoarea: cum ar fi trebuit să trateze această metodă situaţia în care toate datele din fluxul de intrare au fost citite, deci s-a ajuns la sfârşitul fişierului? Ar fi trebuit ca metoda `read()` să arunce o excepţie, pentru că s-a ajuns la finalul fişierului, sau, din contră, această situaţie să nu fie considerată specială, ci mai degrabă una previzibilă? Soluţia aleasă de dezvoltatorii limbajului Java a fost că metoda *nu* trebuie să arunce o excepţie, ci doar să returneze o valoare specială, `-1`, care să indice sfârşitul fişierului. Astfel, atingerea sfârşitului fişierului a fost considerată drept

Digitized by Google

o situaţia normală, previzibilă în utilizarea metodei `read()`. Motivul pentru care nu a fost considerată o situaţie anormală a fost acela că metoda obişnuită de a citi `bytes` dintr-un flux de intrare, este aceea de a încerca citirea lor pe rând, până când este atins finalul fişierului.

Al doilea caz este cel al metodei `readInt()`, clasa `DataInputStream`, din acelaşi pachet `java.io`. În momentul în care este apelată, această metodă citeşte patru `bytes` din fluxul de intrare şi îi converteşte într-un `int`. În această situaţie, creatorii limbajului Java au ales ca metoda `readInt()` să arunce o excepţie, `EOFException`, atunci când se ajunge la sfârşitul unui fişier. Două au fost motivele pentru care această situaţie a fost considerată anormală:

- metoda `readInt()` nu poate returna o valoarea specială pentru a semnala sfârşitul fişierului, pentru că toate valorile pe care le-ar putea returna sunt valori întregi care ar putea exista în acel fişier (cu alte cuvinte, `-1` nu ar putea fi considerat sfârşitul fişierului, pentru că ar putea exista un element `-1` în cadrul fişierului, şi astfel am fi induşi în eroare, deoarece prezenţa elementului `-1` în cadrul acelui fişier nu înseamnă obligatoriu că fişierul a fost parcurs până la final);

- metoda `readInt()` poate fi pusă în ipostaza de a putea citi doar unul, doi sau trei `bytes` din fluxul de intrare, şi nu patru, cât îi este necesar pentru a forma un `int` (o variabilă de tip întreg este reprezentată în Java pe 4 octeţi), ceea ce poate fi considerată o situaţie anormală. Datorită acestui fapt, excepţia `EOFException` este definită ca o excepţie tratată, motiv pentru care programatorii care utilizează metoda `readInt()` sunt obligaţi să prindă această excepţie şi să o trateze.

Al treilea exemplu se referă la interfaţa `java.util.Enumeration`, împreună cu cele două metode ale sale, `hasMoreElements()` şi `nextElement()`. Această interfaţă reprezintă o modalitate de a parcurge o serie de elemente, câte unul pe rând, de la primul element până la ultimul element al seriei. Obţinerea unui element din serie se face cu ajutorul metodei `nextElement()`, care returnează elementul curent din serie şi mută poziţia cursorului pe următorul element, care astfel devine noul element curent. Metoda `hasMoreElements()` returnează o valoare booleană, care indică dacă seria a fost parcursă în întregime sau nu s-a atins încă sfârşitul ei. Metoda `hasMoreElements()` trebuie să fie apelată de fiecare dată înaintea metodei `nextElement()` pentru a vedea dacă s-a ajuns la sfârşitul seriei de elemente sau nu. Această abordare ne arată că dezvoltatorii limbajului Java nu au considerat atingerea sfârşitului seriei de elemente ca fiind o situaţie anormală, ci una previzibilă. Totuşi, dacă sfârşitul seriei a fost atins şi este apelată metoda `nextElement()`, fără a apela

Digitized by Google

metoda `hasMoreElements()` în prealabil, atunci este aruncată o excepţie, `NoSuchElementException`, care reprezintă o excepţie netratată (subclasă a clasei `RuntimeException`), aruncată mai degrabă pentru a indica o eroare de programare (faptul că interfaţa nu este utilizată corect), decât pentru a indica sfârşitul seriei de elemente.

După cum se poate observa, regula general valabilă este că în cazul în care situaţia este previzibilă (de exemplu, sfârşitul unui fişier, al unei serii de elemente, sau obţinerea unui rezultat nul la operaţia de căutare a unui element într-un şir de elemente), *nu* este recomandată utilizarea excepţiilor.

O altă abordare a problemei utilizării eficiente a excepţiilor este legată de modul în care metodele sunt utilizate. Să luăm cazul metodei `chartAt()` a clasei `String`, din pachetul `java.lang`. Această metodă primeşte ca parametru un întreg şi returnează caracterul din string aflat pe poziţia indicată de parametru.

Dacă metoda `charAt()` este apelată cu parametrul `-1` sau un întreg de o valoare mai mare decât lungimea şirului, ea nu îşi poate executa sarcina pentru că datele sale de intrare sunt incorecte. Iar, dacă datele de intrare sunt incorecte, nici cele de ieşire (rezultatele) nu pot fi corecte. De aceea, metoda aruncă o excepţie `StringIndexOutOfBoundsException`, care indică faptul că programatorul are o eroare de programare sau nu a utilizat clasa corect.

Pe de altă parte, dacă datele de intrare sunt corecte, dar dintr-un anumit motiv `chartAt()` nu ar putea returna caracterul de pe poziţia specificată, atunci metoda ar trebui să semnaleze acest lucru aruncând o excepţie, prin care indică faptul că sunt probleme care necesită o tratare specifică.

Prin intermediul acestei abordări se poate trage concluzia că excepţiile trebuie utilizate atunci când metodele nu beneficiază de date de intrare corecte (precondiţie încălcată) sau când, deşi au date de intrare corecte, totuşi, din anumite motive, nu îşi pot duce la bun sfârşit execuţia (postcondiţie încălcată).

Cea de a doua problemă majoră cu care se confruntă programatorii Java este *ce* excepţii să arunce.

După ce s-a decis utilizarea excepţiilor, următorul pas este să decidem ce tipuri de excepţii să utilizăm. Avem de ales între a arunca excepţii instanţe ale clasei `Throwable` sau ale unei subclase a acesteia. Pe de altă parte, se pot arunca excepţii predefinite de platforma Java (existente în Java API), sau putem crea propria ierarhie de excepţii.

Prima sugestie în alegerea tipului de excepţii este aceea de a arunca întotdeauna *excepţii* (subclase ale clasei `Exception`) şi nu *erori* (subclase ale clasei `Error`). Motivul este că `Error` are scopul de a semnala probleme complexe, delicate ale sistemului unde este preferabil să nu ne băgăm nasul.

A doua sugestie este legată de modul de utilizare a excepţiilor tratate (checked

Digitized by Google

exceptions) şi a celor netratate (unchecked exceptions). După cum s-a observat de-a lungul acestui capitol, aruncarea în cadrul unei metode a unei excepţii tratate, forţează codul care apelează metoda respectivă să trateze acea excepţie. Pe de altă parte, aruncarea în cadrul unei metode a unei excepţii netratate (excepţii *runtime*), nu obligă codul care apelează metoda respectivă să trateze acea excepţie, lăsând la latitudinea programatorului decizia de a prinde sau nu excepţia respectivă.

Sugestia în această situaţie este că, dacă sunteţi de părere că excepţia trebuie obligatoriu tratată de utilizatorul codului dumneavoastră, atunci aruncaţi excepţii tratate.

În general, excepţiile care indică o utilizare improprie a metodelor unei clase (o eroare de programare) sunt excepţii netratate. Un exemplu potrivit este cel anterior, în care metoda `charAt()` aruncă excepţia `StringIndexOutOfBoundsException`. Dacă metoda `charAt()` este apelată cu argumentul `-1`, ea aruncă o excepţie prin care semnalează că nu a fost utilizată corect (nu există indicele `-1` într-un şir de elemente). Pe de altă parte, exemplul metodei `readInt()` din clasa `DataInputStream` şi al excepţiei `EOFException` este util pentru a ilustra alegerea excepţiilor tratate. Deoarece excepţia `EOFException` indică o eroare care a survenit în momentul citirii din fişier, nu poate fi vorba de o utilizare greşită a clasei `DataInputStream`. Excepţia semnalează că, deşi clasa a fost utilizată corect, totuşi metoda nu a reuşit să îşi îndeplinească funcţionalitatea, ceea ce reprezintă o situaţie anormală, motiv pentru care programatorii care utilizează această metodă sunt obligaţi să trateze cazul de excepţie.

Cea de-a treia şi ultima sugestie este aceea de a înlocui clasele de excepţii predefinite de platforma Java cu propriile clase de excepţii, atunci când consideraţi că ar fi mai explicit astfel. Puteţi realiza acest lucru dacă prindeţi excepţia predefinită Java, şi, în codul de tratare a excepţiei respective, aruncaţi propria dumneavoastră excepţie.

Utilizarea excepţiilor îşi are propriul ei preţ din punct de vedere al resurselor utilizate. Datorită stivei de apel (call stack), creată de maşina virtuală JVM pentru a reţine ordinea în care metodele au fost apelate, oprirea bruscă din execuţie a unei metode (în caz de excepţie) este mult mai costisitoare decât o execuţie fără incidente a metodei respective, deoarece maşina virtuală consumă resurse căutând în stiva de apel, metoda care conţine secvenţa de cod ce tratează respectiva excepţie. Din acest motiv, excepţiile trebuie utilizate ponderat, mai ales în metodele care se apelează foarte frecvent şi a căror viteză de execuţie este esenţială.

Digitized by Google

6.7 Avantajele utilizării excepţiilor. Concluzii

Existenţa unui mecanism de tratare a erorilor este una dintre principalele modalităţi de a creşte robusteţea unei aplicaţii. Tratarea erorilor este un principiu fundamental pentru orice aplicaţie, în special în Java, unde un ţel foarte important este acela de a crea componente de aplicaţii care vor fi utilizate de alţi programatori. Pentru a asigura dezvoltarea unei aplicaţii profesionale, fiecare dintre aceste componente trebuie să fie robustă.

Excepţiile oferă multe avantaje programatorilor. În primul rând, ele separă codul care tratează erorile de codul care funcţionează normal. Codul suspect de a se executa cu eroare (chiar dacă această eroare poate apare doar în `0,001%` din cazuri) este încadrat într–un bloc `try` pentru a fi ţinut sub observaţie. Chiar dacă utilizarea excepţiilor vă poate ajuta să faceţi codul mai lizibil, prin separarea codului de tratare a erorilor de cel normal, totuşi, utilizarea inadecvată a lor produce un cod dificil de citit şi înţeles. Pe de altă parte, mecanismul Java de tratare a erorilor permite propagarea erorii, prin aruncarea mai departe a erorilor prinse, obligând altă metodă să le trateze. Nu în ultimul rând, verificarea de către compilator a existenţei codului de tratare a unei excepţii tratate, este un prim pas spre a avea un cod stabil şi robust.

Din aceste motive este indicat ca pentru fiecare aplicaţie să se dezvolte o strategie de tratare a excepţiilor. De asemenea, excepţiile nu trebuie să fie considerate ca un adaos la o aplicaţie, ci ca o parte integrantă a acesteia, iar pentru a realiza acest lucru este necesar ca strategia de tratare a excepţiilor să fie adoptată în timpul fazei de proiectare a aplicaţiei, şi nu în faza de implementare.

Rezumat

Capitolul de faţă a prezentat modalitatea în care Java tratează situaţiile de excepţie în cadrul unei aplicaţii.

Excepţiile sunt folosite pentru a semnala situaţii de eroare. O excepţie este generată de clauza `throw` şi se propagă până este prinsă şi tratată de un bloc `catch`, asociat cu un bloc `try`. Pe de altă parte, metodele specifică în cadrul unei liste `throws` excepţiile tratate pe care le aruncă.

Capitolul următor prezintă pe scurt sistemul de intrare/ieşire în limbajul Java.

Noţiuni fundamentale

`catch`: bloc de instrucţiuni utilizat pentru tratarea unei excepţii.

Digitized by Google

excepţie runtime: excepţie netratată, care nu necesită să fie prinsă şi tratată (ex. `NullPointerException`, `IndexOutOfBoundsException`).

excepţie tratată: excepţie care trebuie prinsă sau propagată explicit într-o clauză `throws`.

`finally`: bloc de instrucţiuni care se execută indiferent de rezultatul execuţiei blocului `try-catch`.

`NullPointerException`: tip de excepţie generată de încercarea de a apela un atribut sau o metodă pe o referinţă `null`.

`throw`: instrucţiune utilizată pentru a arunca o excepţie.

`throws`: indică faptul că o metodă poate propaga o excepţie.

`try`: cuprinde codul suspect, care ar putea genera o excepţie.

Erori frecvente

1. Excepţiile tratate trebuie să fie în mod obligatoriu prinse sau propagate în mod explicit prin intermediul clauzei `throws`.

2. Deşi este tentant (de dragul simplităţii) să scriem clauze `catch` vide, aceasta face ca programul să fie foarte dificil de depanat, deoarece excepţiile sunt prinse fără a se semnala deloc acest lucru.

Exerciţii

Pe scurt

1. Descrieţi mecanismul de funcţionare a excepţiilor în limbajul Java.

2. Care excepţii trebuie tratate obligatoriu şi care nu?

În practică

1. Creaţi o metodă `main()` care aruncă un obiect de tip `Exception` în interiorul unui bloc `try`. Furnizaţi constructorului obiectului `Exception` un mesaj de tip `String`. Prindeţi excepţia în cadrul unei clauze `catch` şi afişaţi mesajul de tip `String`. Adăugaţi o clauză `finally` în care să tipăriţi un mesaj pentru a verifica faptul că aţi trecut pe acolo.

2. Creaţi o clasă proprie de excepţii, asemănătoare lui `TeaTooHotException` din paragraful 6.3. Adăugaţi clasei un constuctor cu un parametru de tip `String` care reţine parametrul într-un atribut al clasei. Scrieţi

Digitized by Google

o metodă care afişează `String`-ul stocat. Exersaţi noua clasă creată folosind un bloc `try-catch`.

3. Scrieţi o clasă cu o metodă care să arunce o excepţie de tipul celei descrise la exerciţiul anterior. Încercaţi să o compilaţi fără a preciza excepţia în clauza `throws`. Adăugaţi apoi clauza `throws` adecvată şi apelaţi metoda în cadrul unei clauze `try-catch`.

4. Declaraţi o referinţă şi iniţializaţi-o cu `null`. Apelaţi apoi o metodă pe referinţa respectivă, pentru a verifica faptul că veţi obţine `NullPointer-Exception`. Încadraţi apoi apelul într-un bloc `try-catch` pentru a prinde excepţia.

5. Scrieţi o secvenţă de cod care să genereze şi să prindă o excepţie de tipul `ArrayIndexOutOfBoundsException` sau de tipul `String-IndexOutOfBoundsException`.

6. Definiţi o clasă cu două metode, `f()` şi `g()`. În metoda `f()` aruncaţi o excepţie de un tip definit de dumneavoastră. În metoda `g()` apelaţi metoda `f()`, prindeţi excepţia aruncată de `f()` şi aruncaţi mai departe o excepţie de tip diferit. Testaţi codul în metoda `main()` a clasei.

7. Definiţi o metodă care aruncă (în clauza `throws`) mai multe tipuri de excepţii definite de dumneavoastră. Apelaţi apoi metoda şi prindeţi toate excepţiile aruncate folosind o singură clauză `catch`. Modificaţi apoi programul astfel încât clauza `catch` să prindă doar aceste tipuri de excepţii.

8. Pentru exemplul cu clasa `Shape` din capitolul anterior, modificaţi metodele `readShape()` şi `main()` astfel încât ele să arunce şi să prindă excepţii (în loc să creeze un cerc de rază 0) atunci când se observă o eroare la citire.

9. În multe situaţii o metodă care aruncă o excepţie este reapelată până când execuţia ei decurge normal sau s-a atins un număr maxim de încercări. Creaţi un exemplu care să simuleze acest comportament, prin apelarea unei metode în cadrul unui ciclu `while`, care ciclează până când nu se mai aruncă nici o excepţie.

Digitized by Google

7. Intrare şi ieşire (Java I/O)

"Poţi să îmi spui, te rog, pe unde să ies de aici?"
"Aceasta depinde în mare măsură de unde vrei să mergi", spuse pisica.
"Nu prea îmi pasă unde...", spuse Alice.
"Atunci nu contează pe ce drum ieşi", spuse pisica.

Lewis Caroll, Alice în ţara minunilor

În capitolul anterior am prezentat excepţiile Java şi modalitatea în care se tratează situaţiile de eroare apărute în cadrul unei aplicaţii. În capitolul de faţă vom parcurge pe scurt sistemul Java de intrare-ieşire.

Vom afla detalii despre:

- Cum este structurat sistemul Java de intrare şi ieşire (I/O);
- Care sunt modalităţile de a citi/scrie date într-o aplicaţie Java;
- Cum sunt utilizate pachetele de resurse pentru a simplifica citirea datelor de intrare într-un program Java.

7.1 Preliminarii

Intrarea şi ieşirea în Java se realizează cu ajutorul claselor din pachetul predefinit `java.io`. Din acest motiv, orice program care foloseşte rutinele de intrare/ieşire trebuie să cuprindă instrucţiunea:

Digitized by Google

```
import java.io.*;
```

Biblioteca Java de intrare/ieşire este extrem de sofisticată şi are un număr foarte mare de opţiuni. În cadrul acestui capitol vom examina doar elementele de bază referitoare la intrare şi ieşire, concentrându-ne în întregime asupra operaţiilor de intrare-ieşire formatate (nu vom vorbi despre operaţii de intrare-ieşire binare).

Deşi în practică programele Java îşi preiau datele de intrare din fişiere sau chiar prin Internet, pentru cei care învaţă acest limbaj este uneori mai convenabil să preia datele de la tastatură, aşa cum obişnuiau în C sau Pascal. Vom arăta în paragrafele care urmează cum se pot prelua datele de la tastatură şi cât de puţine modificări sunt necesare pentru a prelua aceleaşi date din fişiere.

7.2 Operaţii de bază pe fluxuri (stream-uri)

Java oferă trei fluxuri predefinite pentru operaţii I/O standard:

- `System.in` (intrarea standard);
- `System.out` (ieşirea standard);
- `System.err` (fluxul de erori).

Aşa cum am arătat deja, metodele `print` şi `println` sunt folosite pentru afişarea formatată. Orice fel de obiect poate fi convertit la o formă tipăribilă, folosind metoda `toString()`; în multe cazuri, acest lucru este realizat automat. Spre deosebire de C şi C++ care dispun de un număr enorm de opţiuni de formatare, afişarea în Java se face exclusiv prin concatenare de `String`-uri, fără nici o formatare.

O modalitate simplă de a realiza citirea este aceea de a citi o singură linie într-un obiect de tip `String` folosind metoda `readLine()`. `readLine()` preia caractere de la intrare până când întâlneşte un terminator de linie sau sfârşit de fişier. Metoda returnează caracterele citite (din care extrage terminatorul de linie) ca un `String` nou construit. Pentru a citi date de la tastatură folosind `readLine()` trebuie să construim un obiect `BufferedReader` dintr-un obiect `InputStreamReader` care la rândul său este construit din `System.in`. Acest lucru a fost ilustrat în capitolul 3, **Listing 3.2**.

Dacă primul caracter citit este `EOF`, atunci metoda `readLine()` returnează `null`, iar dacă apare o eroare la citire, se generează o excepţie de tipul `IOException`. În cazul în care şirul citit nu poate fi convertit la o valoare întreagă, se generează o excepţie `NumberFormatException`.

Digitized by Google

7.3 Obiectul `StringTokenizer`

Obiectul `StringTokenizer` ne permite să separăm elementele lexicale distincte din cadrul unui string. Să ne reamintim că pentru a citi o valoare primitivă, cum ar fi `int`, foloseam `readLine()` pentru a prelua linia ca pe un `String` şi apoi aplicam metoda `parseInt()` pentru a converti stringul la tipul primitiv.

În multe situaţii avem mai multe elemente pe aceeaşi linie. Să presupunem, de exemplu, că fiecare linie are două valori întregi. Putem accesa fiecare dintre aceste două valori, utilizând clasa `StringTokenizer` pentru a separa `String`-ul în elemente lexicale (engl. token). Pentru a putea folosi obiecte ale clasei `StringTokenizer` se foloseşte directiva:

```
import java.util.StringTokenizer;
```

Folosirea obiectului `StringTolenizer` este exemplificată în programul din **Listing 7.1**. Obiectul `StringTokenizer` este construit în linia 19 prin furnizarea obiectului `String` reprezentând linia citită. Apelul metodei `countTokens()` din linia 20 ne va da numărul de cuvinte citite (elemente lexicale). Metoda `nextToken()` întoarce următorul cuvânt necitit ca pe un `String`. Această ultimă metodă generează o excepţie `NoSuchElementException` dacă nu există nici un cuvânt rămas, dar aceasta este o excepţie de execuţie standard şi nu trebuie neapărat prinsă. La liniile 26 şi 27 folosim `nextToken()` urmată de `parseInt()` pentru a obţine un `int`. Toate erorile sunt prinse în blocul `catch`.

Implicit, Java consideră spaţiul, caracterul *tab* sau linia nouă drept delimitator al elementelor lexicale, dar obiectele de tip `StringTokenizer` pot fi construite şi în aşa fel încât să recunoască şi alte caractere drept delimitatori.

Listing 7.1: Program demonstrativ pentru clasa `StringTokenizer`

```
1 //program pentru exemplificarea clasei
2 //StringTokenizer. Programul citeste doua numere aflate pe
3 //aceeasi linie si calculeaza maximul lor
4
5 import java.io.*;
6 import java.util.*;
7
8 public class TokenizerTest
9 {
10   public static void main(String[] args)
11   {
12     BufferedReader in = new BufferedReader(
13     new InputStreamReader(System.in));
14
```


Digitized by Google

```
    System.out.println("Introduceti 2 nr. pe linie:");
    try
    {
      String s = in.readLine();
      StringTokenizer st = new StringTokenizer(s);
      if (st.countTokens() != 2)
      {
        System.out.println("Numar invalid de " +
        "argumente!");
        return;
      }
      int x = Integer.parseInt(st.nextToken());
      int y = Integer.parseInt(st.nextToken());
      System.out.println("Maximul este: " +
      Math.max(x, y));
    }
    catch(Exception e)
    {
      System.out.println(e);
    }
  }
}
```

7.4 Fişiere secvenţiale

Una dintre regulile fundamentale în Java este aceea că ceea ce se poate face pentru operaţii I/O de la terminal se poate face şi pentru operaţii I/O din fişiere. Pentru a lucra cu fişiere, obiectul `BufferedReader` nu se construieşte pe baza unui `InputStreamReader`. Vom folosi în schimb un obiect de tip `FileReader` care va fi construit pe baza unui nume de fişier.

Un exemplu pentru ilustrarea acestor idei este prezentat în **Listing 7.2**. Acest program listează conţinutul fişierelor text care sunt precizate ca parametri în linie de comandă. Funcţia `main()` parcurge pur şi simplu argumentele din linia de comandă, apelând metoda `listFile()` pentru fiecare argument. În metoda `listFile()` construim obiectul de tip `FileReader` la linia 26 şi apoi îl folosim pe acesta pentru construirea obiectului de tip `BufferedReader` în linia următoare. Din acest moment, citirea este identică cu cea de la tastatură.

După ce am încheiat citirea din fişier, trebuie să închidem fişierul, altfel s-ar putea să nu mai putem deschide alte stream-uri (să atingem limita maximă de stream-uri care pot fi deschise). Nu este indicat să facem acest lucru în primul bloc `try`, deoarece o excepţie ar putea genera părăsirea prematură a blocului şi fişierul nu ar fi închis. Din acest motiv, fişierul este închis după secvenţa `try/catch`.

Digitized by Google

Scrierea formatată în fişiere este similară cu citirea formatată.

Listing 7.2: Program care listează conţinutul unui fişier

```
//program pentru afisarea continutului fisierelor
//ale caror nume sunt precizate in linia de comanda
import java.io.* ;

public class Lister
{
  public static void main(String[] args)
  {
    if (args.length == 0)
    {
      System.out.println("Nici un fisier precizat!");
    }

    for (int i = 0 ; i < args.length; ++i)
    {
      listFile(args[i]);
    }
  }

  public static void listFile(String fileName)
  {
    System.out.println("NUME FISIER: " + fileName);

    try
    {
      FileReader file = new FileReader(fileName);
      BufferedReader in = new BufferedReader(file);
      String s;
      while ((s = in.readLine()) != null)
      {
        System.out.println(s);
      }
    }
    catch (Exception e)
    {
      System.out.println(e);
    }
    try
    {
      if (in != null)
      {
        in.close();
      }
    }
    catch (IOException e)
    { //ignora exceptia
    }
```


Digitized by Google

```
    }
}
```

7.5 Utilizarea colecţiilor de resurse (resource bundles)

Vom prezenta acum şi o altă modalitate de a stoca şi accesa date, în care se foloseşte o structurare a datelor la un nivel superior: colecţiile de resurse (resource bundles).

Scopul pentru care colecţiile de resurse au fost implementate de creatorii limbajului Java, nu a fost acela de a le folosi pentru citirea de date, ci pentru a realiza internaţionalizarea aplicaţiilor, cu alte cuvinte pentru a permite traducerea interfeţei unei aplicaţii dintr-o limbă în alta, într-un mod foarte simplu. Utilizarea lor a fost însă extinsă de programatori şi pentru a stoca şi citi date într-un format mai organizat, superior orientării pe octeţi a fluxurilor de date. Vom prezenta în continuare un exemplu simplu de utilizare a colecţiilor de resurse pentru a extrage informaţii dintr-un fişier de proprietăţi. Anexa D prezintă în detaliu modul în care se utilizează colecţiile de resurse şi în alte direcţii.

7.5.1 Utilizarea fişierelor de proprietăţi pentru citirea datelor

Colecţiile de resurse pot fi reprezentate sub mai multe forme în Java. Anexa D vă stă la dispoziţie pentru lămuriri în această privinţă. Ca exemplu, am ales pentru reprezentarea colecţiilor de resurse varianta *fişierelor de proprietăţi*. Fişierele de proprietăţi sunt simple fişiere text, cu extensia `.properties`. Iată un exemplu:

```
#fisierul de proprietati pentru programul Human.java
age=14
height=1.70
```

După cum se poate observa, fişierul de proprietăţi este format din perechi de tipul cheie/valoare. Cheia este de partea stângă a semnului egal, iar valoarea este de partea dreaptă. De exemplu, `age` este cheia care corespunde valorii `14`. Analog, putem spune că `14` este valoarea care corespunde cheii `age`. Cheia este arbitrară. Am fi putut denumi cheia `age` altfel, de exemplu `myAge` sau `a`. Odată definită însă, cheia nu trebuie schimbată pentru că ea este referită în codul sursă. Pe de altă parte, valoarea asociată cheii poate fi schimbată fără probleme.

Digitized by Google

De asemenea, se remarcă faptul că semnul # este folosit pentru comentarea unei linii din cadrul fişierului de proprietăţi.

Să considerăm că fişierul de proprietăţi prezentat mai sus este salvat pe disc în fişierul cu numele `humanProp.properties`. Evident, puteţi schimba acest nume cu unul la alegerea dumneavoastră. Programul Java din **Listing 7.3** prezintă o modalitate de utilizare a acestui pachet de resurse (pentru o execuţie corectă, salvaţi acest fişier cu numele `Human.java` în acelaşi director cu fişierul `humanProp.properties`).

Listing 7.3: Program de test pentru fişiere de proprietăţi

```
import java.util.*;

public class Human
{
    public static void main(String args[])
    {
        try
        {
            ResourceBundle rb = ResourceBundle.getBundle(
                                  "humanProp");

            String s1 = rb.getString("age");
            System.out.println(s1);

            String s2 = rb.getString("height");
            System.out.println(s2);
        }
        catch(Exception e)
        {
            System.out.println("EXCEPTION: " + e.getMessage());
        }
    }
}
```

Utilizarea fişierelor de proprietăţi este, după cum se vede şi în exemplul precendent, destul de simplă. Metoda `getBundle()` realizează accesul la fişierul `humanProp.properties`. Se observă faptul că nu este necesară extensia `.properties` în apelul metodei.

Accesul la valorile cheilor se face prin metoda `getString()`. Trebuie reţinut faptul că toate valorile care sunt preluate din fişierele `properties` sunt de tipul `String`. De aceea, ele trebuie convertite către tipurile lor. În cazul nostru, valoarea cheii `age` trebuie convertită la `int`, iar cea a cheii `height` la `float` sau `double`. Pentru a păstra simplitatea programului, am ales să nu mai convertim aceste valori, ci doar le afişăm pentru a testa valoarea lor. Rezultatul afişării este:

Digitized by Google

```
14
1.70
```

Această modalitate de organizare şi citire a datelor este deosebit de utilă pentru a păstra date a căror organizare logică este de forma cheie=valoare, cum ar fi de exemplu stocarea opţiunilor utilizatorului unei aplicaţii. În cazul unui editor de texte scris în Java, fişierul de proprietăţi ar putea stoca preferinţele utilizatorului referitoare la dimensiunea fontului, culori, directorul de pornire etc.

Rezumat

Ceea ce am prezentat în acest capitol a constituit o prezentare sumară, dar suficientă pentru a putea merge mai departe, a sistemului de intrare-ieşire existent în Java. Au fost prezentate fluxurile de intrare/ieşire, clasa `StringTokenizer`, fişierele secvenţiale şi colecţiile de resurse. Deşi introduse pentru a permite internaţionalizarea aplicaţiilor (vezi anexa dedicată acestei probleme), am văzut cum pot fi utilizate colecţiile de resurse pentru a prelua date din fişiere de proprietăţi.

Noţiuni fundamentale

argument în linie de comandă: parametru scris în linia de comandă în care se realizează execuţia programului respectiv. Acest parametru poate fi folosit în cadrul funcţiei `main` a aplicaţiei rulate.

`BufferedReader`: clasă Java predefinită utilizată pentru citirea pe linii separate a datelor de intrare.

colecţie de resurse: mulţime de proprietăţi care au asociate anumite valori. Cunoscută şi sub numele de pachet de resurse.

`FileReader`: clasă utilizată pentru lucrul cu fişiere.

fişiere de proprietăţi: fişiere text având extensia `.properties`, cu un conţinut special, format din perechi de tipul `proprietate=valoare`, stocate câte una pe linie.

`StringTokenizer`: clasă utilizată pentru a delimita cuvintele existente în cadrul unei propoziţii. Cuvintele sunt despărţite prin separatori.

`System.out, System.in, System.err`: fluxurile standard de ieşire, intrare, eroare pentru operaţii de citire-scriere.

Digitized by Google

Erori frecvente

1. Fişierele de proprietăţi trebuie să aibă extensia `.properties`.

Exerciţii

1. Scrieţi un program care afişează numărul de caractere, cuvinte, linii pentru fişiere precizate în linie de comandă.

2. Implementaţi un program care realizează copierea unui fişier dintr-o locaţie în alta.

Digitized by Google

8. Fire de execuţie

Timpul este modalitatea prin care natura are grijă să nu se producă totul simultan.

Autor anonim

Programarea concurentă reprezintă pentru mulţi programatori Java un concept neobişnuit, care necesită timp pentru a fi asimilat. Efortul necesar pentru tranziţia de la programarea neconcurentă la programarea concurentă este asemănător cu cel pentru tranziţia de la programarea procedurală la programarea orientată pe obiecte: o tranziţie dificilă, frustrantă uneori, dar, în final, efortul este recompensat pe deplin. Dacă la prima parcurgere nu înţelegeţi anumite secvenţe din acest capitol, pentru a vă uşura sufletul daţi vina pe slaba inspiraţie a autorilor, lăsaţi ceva timp pentru ca informaţiile să se asimileze şi încercaţi să rulaţi exemplele, după care reluaţi secvenţa care v-a creat dificultăţi.

În acest capitol vom prezenta:

- Ce este un fir de execuţie;
- Cum se creează aplicaţii care rulează pe mai mult fire de execuţie;
- Cum se evită inconsistenţa la concurenţă folosind metode sincronizate şi instrucţiunea `synchronized;`
- Cum funcţionează monitoarele în Java;
- Cum se coordonează firele de execuţie folosind metodele `wait()` şi `notify()`.

Digitized by Google

8.1 Ce este un fir de execuţie?

Firele de execuţie (*engl.* threads) permit unui program să execute mai multe părţi din el însuşi în acelaşi timp. De exemplu, o parte a programului poate să realizeze o conexiune la un server de internet şi să descarce nişte date, în timp ce cealaltă parte a programului afişează fereastra principală şi meniul aplicaţiei pentru a putea interacţiona în acest timp cu utilizatorul. Un alt exemplu clasic de utilizare a firelor de execuţie îl constituie un browser de Web care vă permite să descărcaţi un număr nedefinit de fişiere (fiecare operaţie de descărcare reprezintă un fir de execuţie), timp în care puteţi să navigaţi nestingheriţi prin alte pagini (pe un alt fir de execuţie).

Firele de execuţie sunt o invenţie relativ recentă în lumea informaticii, deşi *procesele*, surorile lor mai mari, sunt folosite deja de câteva decenii. Firele de execuţie sunt asemănătoare cu procesele, în sensul că pot fi executate independent şi simultan, dar diferă prin faptul că nu implică un consum de resurse atât de mare. Cei care aţi programat în C sau C++ vă amintiţi probabil faptul că prin comanda *fork* se putea crea un nou proces. Un proces astfel creat este o copie *exactă* a procesului original, având aceleaşi variabile, cod etc. Copierea tuturor acestor date face ca această comandă să fie costisitoare ca timp de execuţie şi resurse folosite. În timp ce rulează, procesele generate prin *fork* sunt complet independente de procesul care le-a creat, ele putându-şi chiar modifica variabilele fără a afecta procesul părinte.

Spre deosebire de procese¸ firele de execuţie nu fac o copie a întregului proces părinte, ci pur şi simplu rulează o altă secvenţă de cod în acelaşi timp. Aceasta înseamnă că firele de execuţie pot fi pornite rapid, deoarece ele nu necesită copierea de date de la procesul părinte. În consecinţă, firele de execuţie vor partaja între ele datele procesului căruia îi aparţin. Ele vor putea scrie sau citi variabile pe care oricare alt fir de execuţie le poate accesa. Din acest motiv, comunicarea între firele de execuţie este mai facilă, dar poate conduce şi la situaţia în care mai multe fire de execuţie modifică aceleaşi date în acelaşi timp, conferind rezultate imprevizibile.

Firele de execuţie permit o mai bună utilizare a resurselor calculatorului. În situaţia în care maşina virtuală Java (JVM) rulează pe un sistem multiprocesor, firele de execuţie din cadrul aplicaţiei pot să ruleze pe procesoare diferite, realizând astfel un paralelism real. Am arătat deja că există avantaje efective ale utilizării firelor de execuţie şi pe sistemele cu un singur procesor. Folosirea lor permite scrierea de programe mai simple şi mai uşor de înţeles. Un fir de execuţie poate să execute o secvenţă de cod care realizează o operaţie simplă, lăsând restul muncii altor fire de execuţie.

În rezumat, putem spune că un *proces* este un program de sine stătător, care

Digitized by Google

dispune de propriul lui spaţiu de adrese. Un sistem de operare multitasking este capabil să ruleze mai multe procese (programe) în acelaşi timp, oferindu-le periodic un anumit număr de cicluri procesor, lăsând astfel impresia că fiecare proces se execută de unul singur. Un fir de execuţie (thread) reprezintă un singur flux de execuţie din cadrul procesului. Astfel, un singur proces poate să conţină mai multe fire de execuţie care se execută în mod concurent.

8.2 Fire de execuţie în Java

Specificaţiile limbajului Java definesc firele de execuţie (thread-urile) ca făcând parte din bibliotecile Java standard (pachetele `java.*`). Fiecare implementare a limbajului Java trebuie să furnizeze bibliotecile standard, deci implicit să suporte şi fire de execuţie. În consecinţă, cei care dezvoltă aplicaţii Java pot întotdeauna să se bazeze pe faptul că firele de execuţie vor fi prezente pe orice platformă va rula programul. Mai mult, toate bibliotecile standard Java fie sunt *thread-safe* (adică sunt fiabile în cazul utilizării lor simultane de către mai multe fire de execuţie) sau, dacă nu, au măcar un comportament bine definit în această situaţie. De exemplu, se ştie că clasa `Vector` este thread-safe, deci elementele dintr-un vector pot fi modificate de către mai multe fire de execuţie simultan, fără a se pierde consistenţa datelor. Clasa `ArrayList` este echivalentul clasei `Vector` care nu este thread-safe, dar are avantajul de a fi mai rapidă.

8.2.1 Crearea firelor de execuţie

În Java există două variante pentru a crea fire de execuţie: fie se derivează din clasa `Thread`, fie se implementează interfaţa `Runnable`. Ambele variante conduc la acelaşi rezultat. Dacă se implementează interfaţa `Runnable`, putem converti clase deja existente la fire de execuţie fără a fi nevoie să modificăm clasele din care acestea sunt derivate (acesta este practic singurul avantaj adus de implementarea interfeţei `Runnable`).

Derivarea din clasa `Thread`

Cea mai simplă modalitate de a crea un fir de execuţie este să extindem clasa `Thread`, care conţine tot codul necesar pentru a crea şi rula fire de execuţie. Cea mai importantă metodă pentru clasa `Thread` este `run()`. Prin redefinirea acestei metode se precizează instrucţiunile care trebuie executate de firul de

Digitized by Google

Listing 8.1: Exemplu simplu de fire de execuţie create prin derivarea din clasa Thread

```
1 public class SimpleThread extends Thread
2 {
3   private int countDown = 5;
4   private static int threadCount = 0;
5   private int threadNumber;
6
7   public SimpleThread()
8   {
9     threadNumber = ++threadCount;
10    System.out.println("Creez firul nr. " + threadNumber);
11  }
12
13  public void run()
14  {
15    while(true)
16    {
17      System.out.println("Fir nr. " + threadNumber + "(" +
18                         countDown + ")");
19      if(--countDown == 0)
20      {
21        return;
22      }
23    }
24  }
25
26  public static void main(String[] args)
27  {
28    for(int i = 0; i < 3; i++)
29    {
30      new SimpleThread().start();
31    }
32    System.out.println("Firele de executie au fost pornite.");
33  }
34 }
```

execuţie. Practic, metoda `run()` furnizează codul care va fi executat în paralel cu codul celorlalte fire de execuţie din cadrul programului.

Programul din **Listing 8.1** creează 3 fire de execuţie, fiecare fir având asociat un număr unic, generat cu ajutorul variabilei statice `threadCount`. Metoda `run()` este redefinită în liniile 13-24 pentru a scădea valoarea unui contor (variabila `countDown`) la fiecare execuţie a ciclului `while` până când valoarea contorului ajunge la 0, moment în care metoda `run()` se încheie, iar firul de execuţie este terminat.

Digitized by Google

În marea majoritate a cazurilor, instrucţiunile din metoda `run()` sunt cuprinse într-o instrucţiune de ciclare care se execută până când firul de execuţie nu mai este necesar (în cazul nostru, până când contorul ajunge la 0).

În metoda `main()`, liniile 26-32, se crează şi se pornesc 3 fire de execuţie. Metoda `start()` din clasa `Thread` iniţializează firul de execuţie după care îi apelează metoda `run()`. Aşadar, paşii pentru crearea unui fir de execuţie sunt: se apelează constructorul pentru a crea obiectul, apoi metoda `start()` configurează firul de execuţie şi apelează metoda `run()`. Dacă nu se apelează metoda `start()` (lucru care poate fi făcut chiar şi în constructor) firul de execuţie nu va fi pornit niciodată.

Ieşirea afişată de programul din **Listing 8.1** (care *diferă* de la o execuţie la alta) este:

```
Creez firul nr. 1
Creez firul nr. 2
Fir de executie nr. 1(5)
Fir de executie nr. 1(4)
Fir de executie nr. 1(3)
Fir de executie nr. 1(2)
Creez firul nr. 3
Fir de executie nr. 1(1)
Firele de executie au fost pornite.
Fir de executie nr. 3(5)
Fir de executie nr. 3(4)
Fir de executie nr. 3(3)
Fir de executie nr. 2(5)
Fir de executie nr. 3(2)
Fir de executie nr. 3(1)
Fir de executie nr. 2(4)
Fir de executie nr. 2(3)
Fir de executie nr. 2(2)
Fir de executie nr. 2(1)
```

Observaţi că firele de execuţie nu au fost executate în ordinea în care au fost create. De fapt, ordinea în care procesorul rulează firele de execuţie este nedeterminată, cu excepţia situaţiei în care se foloseşte metoda `setPriority()` pentru a modifica prioritatea de execuţie a unui thread.

Un alt lucru demn de remarcat este că la crearea unui fir de execuţie în linia 30, metoda `main()` nu reţine nici o referinţă către obiectele nou create. Un obiect Java obişnuit ar fi în această situaţie disponibil pentru colectorul de

Digitized by Google

gunoaie (care, dacă tot veni vorba, rulează şi el pe un fir de execuţie separat), dar nu şi un fir de execuţie. Fiecare fir de execuţie se înregistrează pe el însuşi în cadrul maşinii virtuale, deci există undeva o referinţă către el, astfel încât colectorul de gunoaie nu va putea să îl distrugă atât timp cât firul este în execuţie.

Implementarea interfeţei `Runnable`

Ajunşi cu lectura în acest punct al cărţii ar trebui să fiţi familiarizaţi cu noţiunea de interfaţă, prezentată în capitolul 5. Implementarea interfeţei `Runnable` reprezintă o modalitate elegantă prin care putem face ca orice clasă care o implementează să ruleze pe un fir de execuţie distinct. Iată codul pentru interfaţa `Runnable`:

```
public interface Runnable extends Object
{
  public abstract void run();
}
```

Aşadar interfaţa `Runnable` este foarte simplă, însă deosebit de puternică. Toate clasele care implementează această interfaţă trebuie, desigur, să implementeze metoda `run()`. Este interesant de remarcat faptul că noul fir de execuţie se va crea tot cu ajutorul clasei `Thread`, care oferă un constructor având ca parametru o instanţă `Runnable`. Clasa `Thread` va executa în această situaţie metoda `run()` a parametrului în cadrul metodei sale `main()`. Firul de execuţie se va încheia în momentul în care metoda `run()` se încheie. Astfel, această interfaţă foarte simplă permite ca oricare clasă care o implementează să ruleze pe un fir de execuţie separat.

Ajunşi în acest punct, v-aţi pus probabil întrebarea: de ce este necesar să avem şi clasa `Thread` şi interfaţa `Runnable`? Există situaţii în care este mai adecvat să se implementeze interfaţa `Runnable` şi situaţii în care este de preferat să se extindă clasa `Thread`. Crearea unui fir de execuţie prin implementarea interfeţei `Runnable` este mai flexibilă, deoarece poate fi întotdeauna utilizată. Nu veţi putea întotdeauna să extindeţi clasa `Thread`. Adevărata putere a interfeţei `Runnable` iese în evidenţă în momentul în care aveţi o clasă care trebuie să extindă o altă clasă pentru a moşteni un anumit gen de funcţionalitate. În acelaşi timp, aţi dori ca acea clasă să ruleze pe un fir de execuţie separat. Aici este cazul să folosiţi `Runnable`.

Odată ce aveţi o clasă care implementează `Runnable` şi defineşte o metodă `run()`, sunteţi pregătiţi pentru a crea un fir de execuţie. Şi în această situaţie va trebui să instanţiaţi clasa `Thread`. Fiecare fir de execuţie în Java este asociat cu o instanţă a clasei `Thread`, chiar dacă se utilizează interfaţa `Runnable`. Aşa cum am amintit deja, clasa `Thread` oferă un constructor care primeşte

Digitized by Google

ca parametru o instanţă `Runnable`. La instanţierea clasei `Thread`, cu acest constructor, metoda `run()` a clasei `Thread` apelează pur şi simplu metoda `run()` a clasei primită ca parametru. Puteţi verifica acest lucru cu uşurinţă, executând următorul program:

```
1 public class TestRunnable implements Runnable
2 {
3   public static void main(String[] args)
4   {
5     new Thread( new TestRunnable() ).start();
6   }
7   public void run()
8   {
9     new Exception().printStackTrace();
10    return ;
11  }
12 }
```

Programul de mai sus reprezintă un exemplu simplu de creare a unui fir de execuţie prin implementarea interfeţei `Runnable`. La linia 1 declarăm faptul că clasa noastră implementează interfaţa `Runnable`. În liniile 7-11 definim metoda `run()` care nu face decât să apeleze metoda `printStackTrace()` a clasei `Exception`, după care se încheie şi, odată cu ea, se încheie şi firul de execuţie. La linia 5 se porneşte firul de execuţie, prin crearea unei instanţe a clasei `Thread` având ca parametru `Runnable` chiar clasa noastră. Rezultatul afişat de program este:

```
java.lang.Exception at
TestRunnable.run(TestRunnable.java:9) at
java.lang.Thread.run(Thread.java:484)
```

ceea ce dovedeşte că metoda `run()` a clasei `TestRunnable` a fost apelată de metoda `run()` a clasei `Thread`.

Programul din **Listing 8.2** reprezintă varianta `Runnable` a programului din **Listing 8.1**. Remarcaţi diferenţele minime între cele două variante. La linia 1, clasa `SimpleRunnable` declară că implementează interfaţa `Runnable`. Apoi, la linia 30 firul de execuţie se creează transmiţând a instanţă a clasei `SimpleRunnable` constructorului clasei `Thread`. În rest, cele două programe sunt identice.

8.3 Accesul concurent la resurse

Un program care rulează pe un singur fir de execuţie poate fi privit ca o unică entitate care se mişcă prin spaţiul problemei făcând un singur lucru la

Digitized by Google

Listing 8.2: Exemplu simplu de fire de execuţie create prin implementarea interfeţei Runnable

```
public class SimpleRunnable implements Runnable
{
  private int countDown = 5;
  private static int threadCount = 0;
  private int threadNumber ;

  public SimpleRunnable()
  {
    threadNumber = ++threadCount ;
    System.out.println("Creez firul nr. " + threadNumber);
  }

  public void run()
  {
    while(true)
    {
      System.out.println("Fir nr. " + threadNumber + "(" +
                         countDown + ")");
      if(--countDown == 0)
      {
        return;
      }
    }
  }

  public static void main(String[] args)
  {
    for(int i = 0; i < 3; i++)
    {
      new Thread( new SimpleRunnable() ).start();
    }
    System.out.println("Firele de executie au fost pornite.");
  }
}
```


Digitized by Google

un moment dat. Deoarece există doar o singură entitate nu este necesar să ne punem problema dacă două entităţi ar putea accesa aceeaşi resursă în acelaşi timp, cum ar fi de exemplu situaţia a două persoane care îşi parchează maşina în acelaşi loc simultan, sau care încearcă să iasă în acelaşi timp pe uşă, sau chiar să vorbească în acelaşi timp.

În cazul mai multor fire de execuţie, nu mai avem o singură entitate şi există deci posibilitatea ca două sau mai multe fire de execuţie să încerce să utilizeze aceeaşi resursă în acelaşi timp. Nu întotdeauna accesarea unei resurse simultan este o problemă, dar sunt situaţii în care aceste coleziuni trebuie prevenite (de exemplu, două fire de execuţie care accesează simultan acelaşi cont în bancă, sau care doresc să tipărească la aceeaşi imprimantă etc.).

Exemplul simplu de mai jos pune în evidenţă problemele care pot să apară într-un program multithreaded (care rulează pe mai multe fire de execuţie):

```
public class Counter
{
  private int count = 0 ; //counter incrementat de increment()
  public int increment()
  {
    int aux = count ;
    count++ ;
    return aux ;
  }
}
```

Clasa de mai sus este extrem de simplă, având un singur atribut şi o singură metodă. Aşa cum îi sugerează şi numele, clasa `Counter` este utilizată pentru a număra diverse lucruri, cum ar fi numărul de apăsări ale unui buton, sau numărul de accesări al unei pagini web. Esenţa clasei este metoda `increment()` care incrementează şi returnează valoarea curentă a counter-ului. Deşi clasa `Counter` pare complet inofensivă, ea consituie o potenţială sursă de comportament imprevizibil într-un mediu multithreaded.

Să considerăm ca exemplu un program Java care conţine două fire de execuţie, amândouă fiind pe cale să execute următoarea linie de cod:

```
int count = counter.increment() ;
```

Programatorul nu poate nici să controleze şi nici să prezică ordinea în care cele două fire de execuţie vor fi executate. Sistemul de operare (sau maşina virtuală Java) deţine controlul complet asupra planificării firelor de execuţie. În consecinţă, nu există nici o certitudine în privinţa cărui fir de execuţie se va executa, sau cât timp va fi executat un anumit fir de execuţie. Orice fir de execuţie poate fi întrerupt printr-o schimbare de context în orice moment. Mai mult, două fire de execuţie pot rula simultan pe două procesoare separate ale unei maşini multiprocesor.

Digitized by Google

Tabela 8.1 descrie o posibilă secvenţă de execuţie a celor două fire de execuţie. În acest scenariu, primul fir de execuţie este lăsat să ruleze până când încheie apelul către metoda `increment()`; apoi, cel de-al doilea fir face acelaşi lucru. Nu este nimic surprinzător la acest scenariu. Primul fir va incrementa valoarea contorului la 1, iar al doilea o va incrementa la 2.

Tabela 8.1: Counter - Scenariul 1

Fir de execuţie 1	Fir de execuţie 2	count	aux
count = counter.increment()	—	0	-
int aux = count //0	—	0	0
count++ //1	—	1	0
return aux //0	—	1	0
—	count = counter.increment()	1	-
—	aux = count //1	1	1
—	count++ //2	2	1
—	return aux //1		1

Tabela 8.2 descrie o secvenţă de execuţie oarecum diferită. În această situaţie, primul fir de execuţie este întrerupt de o schimbare de context chiar în timpul execuţiei metodei `increment()`. Primul fir este astfel suspendat temporar şi se permite execuţia celui de-al doilea. Al doilea fir execută apelul către metoda `increment()`, mărind valoarea contorului la 1 şi returnând valoarea 0. Când primul fir îşi va relua execuţia, problema devine deja evidentă. Valoarea contorului este crescută la 2, dar valoarea returnată este tot 0!

Tabela 8.2: Counter - Scenariul 2

Fir de execuţie 1	Fir de execuţie 2	count	aux
count = counter.increment()	—	0	-
int aux = count //0	—	0	0
—	count = counter.increment()	0	-
—	aux = count //0	0	0
—	count++ //1	1	0
—	return aux //0	0	0
count++ //2	—	2	0
return aux //0	—	2	0

Examinând scenariul din **Tabela 8.2**, veţi observa o secvenţă interesantă de

Digitized by Google

execuţie a operaţiilor. La intrarea în metoda `increment()`, valoarea atributului `count` (0) este stocată în variabila locală `aux`. Primul fir este apoi suspendat pentru o perioadă, timp în care se execută celălalt fir (este important să observăm că valoarea lui `count` se modifică în această perioadă). Când primul fir îşi reia execuţia, el incrementează corect valoarea lui `count` şi returnează valoarea din `aux`. Problema este că valoarea din `aux` nu mai este corectă, deoarece valoarea lui `count` a fost modificată între timp de către celălalt fir. Pentru a ne convinge de faptul că scenariul din **Tabela 8.2** este într-adevăr posibil, am extins clasa `Counter` prezentată anterior, adăugându-i clasa interioară (vezi paragraful 5.7, pagina 145) `CounterTest` (**Listing 8.3**) derivată din `Thread`. Metoda `run()` a clasei `CounterTest` (liniile 34-42) apelează într-un ciclu infinit metoda `increment()` a clasei `Counter`. Variabilele `check1` şi `check2` declară şi iniţializează câte un obiect de tip `CounterTest` care rulează pe un fir de execuţie separat (firul de execuţie este pornit chiar în constructorul clasei `CounterTest` care apelează metoda `start()` în linia 30).

Listing 8.3: Clasă Java care evidenţiază problemele care pot apărea la partajarea resurselor

```
1  public class Counter
2  {
3    private int count = 0; //counter incrementat de increment()
4    //initializeaza si porneste primul thread de verificare
5    private CounterTest check1 = new CounterTest();
6    //initializeaza si porneste al doilea thread de verificare
7    private CounterTest check2 = new CounterTest();
8
9    public int increment()
10   {
11     int aux = count;
12     count++;
13     if( count != aux+1 )
14     {
15       System.out.println("Oops! Ceva nu e in regula... " +
16                          this.count + " " + aux);
17     }
18     return aux;
19   }
20
21   /**
22    * Clasa care evidentiaza problemele de sincronizare
23    * ale metodei increment.
24    */
25   public class CounterTest extends Thread
26   {
27     /** Construieste si porneste firul de executie.*/
```


Digitized by Google

```
28     public CounterTest()
29     {
30       this.start();
31     }
32
33     /** Apeleaza metoda increment() intr-un ciclu infinit.*/
34     public void run()
35     {
36       System.out.println("Firul " + this.getName() +
37                          " a pornit");
38       for(;;)
39       {
40         increment();
41       }
42     }
43   }
44
45   public static void main( String[] args )
46   {
47     //creaza o instanta a clasei Counter
48     new Counter();
49   }
50 }
```

În mod aparent paradoxal, după secvenţa de instrucţiuni

```
int aux = count;
count++;
```

la linia 13 se verifică dacă valoarea variabilei `count` este diferită de `aux+1`

```
if( count != aux+1 )
```

Acest test ar fi complet inutil într-un program care rulează pe un singur fir de execuţie, dar nu şi în cazul nostru. La rularea programului `Counter.java`, acesta va afişa (valorile afişate diferă, desigur, de la o execuţie la alta):

```
Firul Thread-0 a pornit
Firul Thread-1 a pornit
Oops! Ceva nu e in regula... Thread-1 18777999
18247501
Oops! Ceva nu e in regula... Thread-1 55129926
54599470
Oops! Ceva nu e in regula... Thread-0 81760783
81229276
Oops! Ceva nu e in regula... Thread-0 104069819
103556887
Oops! Ceva nu e in regula... Thread-0 118311248
```


Digitized by Google

```
117779680
Oops! Ceva nu e in regula... Thread-1 279005154
278475302
```

Din cele afişate reiese că testul de la linia 13 nu este deloc inutil. Se poate întâmpla ca unul dintre firele de execuţie ale clasei `CounterTest` (să zicem `check1`) să fie întrerupt imediat după ce a fost atribuită valoarea lui `count` variabilei `aux`, dar înainte ca metoda să apuce să returneze valoarea din `aux` (ca în scenariul din **Tabela 8.2**). În această situaţie, celălalt fir de execuţie al clasei `CounterTest`, `check2`, va continua să incrementeze valoarea lui `count` de un număr nedeterminat de ori, după care controlul va fi din nou preluat de firul de execuţie întrerupt, `check1`. Acesta va avea noua valoare a lui `count`, care a fost între timp incrementată, dar vechea valoare a lui `aux`, care (fiind variabilă locală) a fost salvată pe stivă înainte de schimbarea de context[1], de aici rezultând inconsistenţa semnalată de testul din linia 13. Desigur, şansele ca primul fir de execuţie să fie întrerupt chiar după ce a fost incrementat `count` sunt foarte mici, dovadă şi faptul că această situaţie a apărut după peste 18 milioane de apeluri ale metodei `increment()`. Totuşi, şansele există, şi neluarea lor în considerare poate avea consecinţe grave asupra funcţionării unui program.

Problema pusă în evidenţă de scenariul din **Tabela 8.2** poartă numele de *inconsistenţă la concurenţă* (engl. race condition)- rezultatul unui program depinde de ordinea în care s-au executat firele de execuţie pe procesor. În general, nu este deloc indicat să se permită incosistenţe la concurenţă în rezultatele unui program. Noi nu putem şti niciodată în ce moment un fir de execuţie este executat sau întrerupt. Închipuiţi-vă că sunteţi aşezat la o masă cu o lingură în mână gata-gata să gustaţi dintr-o salată de fructe delicioasă care se află în faţa dumneavoastră. Vă îndreptaţi plini de speranţă lingura către salata de fructe şi, chiar în momentul în care să atingeţi salata, aceasta dispare fără urmă (deoarece firul vostru de execuţie a fost suspendat, şi între timp, alt fir de execuţie a furat mâncarea). Mergând cu analogia mai departe, dumneavoastră nu aveţi timp să reacţionaţi la dispariţia farfuriei (fiecare fir de execuţie are iluzia că execută singur şi continuu pe procesor) şi veţi înfige lingura în masă, provocând astfel îndoirea ei (adeseori, folosirea simultană a unei resurse conduce la coruperea ei).

Există situaţii în care nu are importanţă dacă o resursă este accesată în acelaşi timp de mai multe fire de execuţie (mâncarea este pe o altă farfurie). Totuşi, pentru ca programele *multithreaded* (rom. cu mai multe fire de execuţie)

[1]Deşi nu am precizat clar acest lucru, variabilele locale sunt distincte pentru fiecare fir de execuţie în parte. Totuşi, atributele claselor sunt partajate de către toate firele de execuţie care rulează la un moment dat.

Digitized by Google

să funcţioneze, trebuie să existe un mecanism care să permită împiedicarea accesului la o resursă de către mai multe thread-uri, măcar în timpul perioadelor critice. Să ne imaginăm ce s-ar fi întâmplat dacă metoda `increment()` ar fi fost utilizată pentru a actualiza contul în bancă al unei persoane în loc să incrementeze pe `count`. Pentru o bancă, corectitudinea conturilor este de importanţă maximă. Dacă o bancă face erori de calcul sau raportează informaţii incorecte, cu siguranţă nu îşi va încânta clienţii.

Orice programe multithreaded, chiar şi cele scrise în Java, pot suferi de inconsistenţă la concurenţă. Din fericire, Java ne pune la dispoziţie un mecanism simplu pentru a controla concurenţa: *monitoarele*.

8.3.1 Sincronizare

Există multe lucrări de programare şi sisteme de operare care se ocupă de problema programării concurente. Concurenţa a fost un subiect foarte cercetat în ultimii ani. Au fost propuse şi implementate mai multe mecanisme de control al concurenţei, printre care:

- secţiunile critice;
- semafoare;
- mutexuri;
- blocarea de înregistrări în baze de date;
- monitoare.

Limbajul Java implementează o variantă a monitoarelor (engl. *monitors*) pentru controlul concurenţei. Conceptul de *monitor* a fost introdus de C.A.R. Hoare (acelaşi Hoare care a propus şi algoritmul de partiţionare de la Quicksort) în anul 1974 într-o lucrare publicată în *Communications of the ACM*. Hoare descrie în această lucrare un obiect special, denumit monitor, care este utilizat pentru a furniza excluderea reciprocă pentru un grup de proceduri ("excludere reciprocă" este un mod elegant de a spune "Un singur fir de execuţie la un moment dat"). În modelul propus de Hoare, fiecare grup de proceduri care necesită excludere reciprocă este pus sub controlul unui monitor. În timpul execuţiei, monitorul permite să se execute doar o singură procedură aflată sub controlul lui la un moment dat. Dacă un alt fir de execuţie încearcă să apeleze o procedură controlată de monitor, acel fir este suspendat până în momentul în care firul curent îşi încheie apelul.

Digitized by Google

Monitoarele din Java implementează îndeaproape conceptele iniţiale ale lui Hoare, aducând doar câteva modificări minore (pe care nu le vom prezenta în această carte). Monitoarele din Java impun excluderea reciprocă la accesarea de metode, sau, mai exact, excluderea reciprocă la accesarea de metode sincronizate (vom vedea imediat un exemplu concret de metodă sincronizată).

În momentul în care se apelează o metodă Java sincronizată se declanşează un proces deosebit de laborios. În primul rând, maşina virtuală Java va localiza monitorul asociat cu obiectul a cărui metodă este apelată (de exemplu, dacă se apelează `counter.increment()`, maşina virtuală va localiza monitorul lui `counter`). Orice obiect Java poate să aibă un monitor asociat, deşi, din motive de eficienţă, maşina virtuală crează monitoarele doar atunci când este nevoie de ele. Odată ce monitorul este găsit, maşina virtuală încearcă să atribuie monitorul firului de execuţie care a apelat metoda sincronizată. Dacă monitorul nu este atribuit altui fir de execuţie, se va atribui firului curent, căruia îi este apoi permis să continue cu apelul metodei. În situaţia în care monitorul este deja deţinut de către un alt fir de execuţie, el nu va putea fi atribuit firului curent, care va fi pus în aşteptare până când monitorul va fi eliberat. În momentul în care monitorul va deveni disponibil, aceasta va fi atribuit firului de execuţie curent, care va putea continua cu apelul metodei.

Metaforic vorbind, un monitor acţionează ca un fel de secretară aflată la intrarea în birou a unei persoane foarte solicitate. În momentul în care se apelează o metodă sincronizată, secretara permite firului apelant să treacă, după care închide uşa de la birou. Atâta timp cât firul este în cadrul metodei sincronizate, apelurile către alte metode sincronizate ale obiectului sunt blocate. Aceste fire se aşează la coadă în faţa uşii de la birou, aşteptând cu răbdare ca primul fir să plece. În momentul în care primul fir părăseşte metoda sincronizată, secretara deschide din nou uşa de la birou permiţând unui singur fir din coada de aşteptare să intre pentru a-şi executa metoda sincronizată, după care procesul se repetă din nou.

Mai simplu spus, un monitor Java impune un acces de tipul un singur fir la un moment dat. Această operaţie mai este numită adeseori şi serializare (a nu se confunda cu serializarea obiectelor, care permite să citim şi să scriem obiecte Java într-un *stream* (rom. flux de date). Aici serializare are sensul de a permite accesul succesiv).

Observaţie: Programatorii care sunt deja familiarizaţi cu conceptul de programare multithreaded din alte limbaje de programare, tind adeseori să confunde monitoarele cu secţiunile critice. Declararea unei metode sincronizate nu implică faptul că doar un singur fir de execuţie va executa acea metodă la un moment dat, cum este cazul la secţiunile critice. Ea impune doar ca numai

Digitized by Google

un singur fir de execuţie să poată executa acea metodă (sau oricare altă metodă sincronizată) *pe un anumit obiect* la un moment dat. Monitoarele din Java sunt asociate cu obiecte, şi nu cu linii de cod. Este foarte posibil ca două fire de execuţie să execute simultan aceeaşi metodă sincronizată, cu condiţia ca metoda să fie invocată pe obiecte distincte (adică `a.method()` şi `b.method()`, unde `a!=b`).

Pentru a vedea cum operează monitoarele, să rescriem clasa `Counter`, astfel încât să se folosească de monitoare, prin intermediul cuvântului cheie `synchronized`.

Listing 8.4: Clasa `Counter` cu metoda sincronizată

```
public class CounterSync
{
  private int count = 0 ; //counter incrementat de increment()
  public synchronized int increment()
  {
    int aux = count ;
    count++ ;
    return aux ;
  }
}
```

Observaţi că nu a fost necesar să rescriem metoda `increment()` - aceasta a rămas la fel ca în exemplul anterior, cu deosebirea că metoda a fost declarată a fi `synchronized`.

Ce s-ar întâmpla dacă am folosi clasa `CounterSync` în programul din **Listing 8.3**? Ar mai fi posibilă apariţia scenariului din **Tabela 8.2**, care pune în evidenţă inconsistenţa la concurenţă? Rezultatul aceleiaşi secvenţe de schimbări de context, nu ar mai fi identic, deoarece metoda sincronizată va împiedica inconsistenţa la concurenţă. Scenariul modificat din **Tabela 8.2** este prezentat în **Tabela 8.3**.

Digitized by Google

Tabela 8.3: `Counter` - Scenariul 3

Fir de execuţie 1	Fir de execuţie 2	count	aux
count = counter.increment()	—	0	-
(obţine monitorul)	—	0	-
int aux = count //0	—	0	0
—	count = counter.increment()	0	-
—	(nu poate obţine monitorul)	0	-
count++ //1	— blocat	1	0
return aux //0	— blocat	1	0
(eliberează monitorul)	— blocat	1	0
—	(obţine monitorul)	1	-
—	int aux = count //1	1	1
—	count++ //2	2	1
—	return aux //1	2	1
—	(eliberează monitorul)		

Secvenţa de operaţii din **Tabela 8.3** începe la fel ca cea din scenariul anterior: primul fir de execuţie începe să execute metoda `increment()` a clasei `CounterSync`, după care este întrerupt de o schimbare de context. Totuşi, de data aceasta, în momentul în care al doilea fir de execuţie doreşte să apeleze metoda `increment()` pe acelaşi obiect de tip `CounterSync`, acesta este blocat. Tentativa celui de-al doilea fir de a obţine monitorul obiectului eşuează, deoarece acesta este deja deţinut de primul fir. În momentul în care primul fir eliberează monitorul, al doilea fir îl va putea obţine şi va executa metoda `increment()`. Putem să ne convingem de aceasta, modificând programul din **Listing 8.3** prin adăugarea cuvântului `synchronized` la metoda `increment()`. La execuţie programul va afişa:

```
Firul Thread-0 a pornit
Firul Thread-1 a pornit
```

după care va rula până când va fi oprit cu CTRL-C, fără a mai afişa nici un mesaj.

Cuvântul cheie `synchronized` este singura soluţie pe care limbajul Java o oferă pentru controlul concurenţei. Aşa cum s-a văzut şi în exemplul cu clasa `Counter`, incosistenţa potenţială la concurenţă a fost eliminată prin adăugarea modificatorului `synchronized` la metoda `increment()`. Astfel, toate apelurile metodei `increment()` au fost serializate. În general vorbind, modificatorul `synchronized` ar trebui adăugat la orice metodă care modifică

Digitized by Google

atributele unui obiect. Adeseori este deosebit de dificil să examinăm metodele unei clase pentru a detecta posibile inconsistenţe la concurenţă. Este mult mai comod să marcăm toţi modificatorii ca fiind `synchronized`, rezolvând astfel orice problemă de concurenţă.

Observaţie: Unii programatori mai curioşi s-ar putea întreba când o să vadă şi ei un monitor Java. Totuşi, acest lucru nu este posibil. Monitoarele Java nu dispun de o descriere oficială în specificaţia Java, iar implementarea lor nu este direct vizibilă pentru un programator. Monitoarele nu sunt obiecte Java: ele nu au atribute sau metode. Monitoarele sunt un concept aflat la baza implementării modelului de concurenţă în Java. Se poate accesa monitorul unui obiect la nivel de cod nativ, dar acest lucru nu este recomandat, iar descrierea lui depăşeşte cadrul acestui capitol.

8.3.2 Capcana metodelor nesincronizate

Monitoarele din Java sunt folosite doar în conjuncţie cu metodele `synchronized`[2]. Metodele care nu sunt declarate ca fiind `synchronized` nu încearcă să obţină monitorul obiectului înainte de a fi executate. Ele pur şi simplu ignoră complet monitoarele. La un moment dat, cel mult un fir de execuţie poate să execute o metodă sincronizată a unui obiect, dar un număr arbitrar de fire de execuţie pot executa metode nesincronizate. Acest fapt poate să prindă pe picior greşit mulţi programatori începători, care nu sunt suficient de atenţi la care metode trebuie declarate `synchronized` şi care nu. Să considerăm clasa `Account` din **Listing 8.5**, care descrie un cont în bancă împreună cu operaţiile care se realizează asupra lui.

Listing 8.5: Clasa Account

```
public class Account
{
  private int balance;
  /** Creaza un cont cu soldul balance */
  public Account(int balance)
  {
    this.balance = balance;
  }
  /**
   * Transfera suma amount din contul curent
   * in contul destination
   */
```

[2] Afirmaţia este un pic inexactă. Vom vedea imediat că monitoarele Java sunt folosite şi în cazul instrucţiunii `synchronized()`.

Digitized by Google

```
13   public synchronized void transfer(int amount,
14                                     Account destination)
15   {
16     this.withdraw(amount);
17     destination.deposit(amount);
18   }
19   /** Retrage suma amount din cont */
20   public synchronized void withdraw(int amount)
21   {
22     balance -= amount;
23   }
24   /** Depune suma amount in cont */
25   public synchronized void deposit(int amount)
26   {
27     balance += amount;
28   }
29   /** Returneaza soldul din cont */
30   public int getBalance()
31   {
32     return balance;
33   }
34 }
```

Toţi modificatorii clasei `Account` sunt declaraţi ca fiind `synchronized`. Aparent, această clasă nu ar trebui să aibă nici un fel de probleme legate de inconsistenţa la concurenţă. Şi totuşi, ea are!

Pentru a înţelege inconsistenţa la concurenţă de care suferă clasa `Account`, să vedem cum lucrează o bancă cu conturile. Pentru o bancă este de o importanţă majoră să menţină corectitudinea datelor din conturi. Pentru a evita raportarea de informaţii inconsistente, băncile preferă ca, în timp ce se realizează o tranzacţie, să dezactiveze cererile privitoare la soldul dintr-un cont. Astfel, clienţii băncii nu pot să primească informaţii preluate în timpul unei tranzacţii parţial încheiate. Metoda `getBalance()` din clasa `Account` nu este sincronizată, iar acest fapt poate conduce la anumite probleme.

Să presupunem că un fir de execuţie apelează metoda sincronizată `transfer()`, pentru a trece suma de 10 milioane de lei din contul curent în alt cont. Mai mult, firul nostru de execuţie este întrerupt de o schimbare de context imediat după ce a executat linia 16:

```
this.withdraw(amount);
```

În acest moment, suma de 10 milioane a fost extrasă din primul cont, dar încă nu a fost depozitată în celălalt cont. Dacă un alt fir de execuţie va apela metoda (nesincronizată!) `getBalance()` pentru cele două conturi implicate în transfer, aceasta va putea accesa nestingherită informaţia din cele două conturi. Astfel, `getBalance()` va raporta că suma de 10 milioane de lei a fost

Digitized by Google

extrasă din primul cont, dar această sumă nu se va regăsi în contul destinaţie, deoarece metoda `transfer()` nu a apucat să actualizeze acest cont înainte de a fi întreruptă! Astfel, clienţii s-ar putea întreba unde a dispărut suma respectivă[3]. Dacă `getBalance()` ar fi fost sincronizată, acest scenariu nu ar fi fost posibil, deoarece apelul ei ar fi fost blocat până la încheierea tranzacţiei.

8.3.3 Instrucţiunea `synchronized`

Metodele sincronizate nu pot fi folosite în orice situaţie. De exemplu, şirurile Java nu dispun de metode, cu atât mai puţin de metode sincronizate. Există totuşi multe cazuri în care dorim ca accesul la elementele unui şir să fie sincronizat. Pentru a rezolva această situaţie, limbajul Java oferă şi o altă convenţie sintactică prin care putem interacţiona cu monitorul unui obiect: instrucţiunea `synchronized`, cu următoarea sintaxă:

```
synchronized( expresie )
  instructiune
urmatoarea instructiune
```

Execuţia unei instrucţiuni `synchronized` are acelaşi efect ca şi apelarea unei metode sincronizate: firul de execuţie va trebui să deţină un monitor înainte de a executa `instrucţiune` (care poate fi simplă sau compusă). În cazul instrucţiunii `synchronized` se va utiliza monitorul obiectului obţinut prin evaluarea lui `expresie` (al cărei rezultat nu trebuie să fie un tip primitiv).

Instrucţiunea `synchronized` este cel mai adesea folosită pentru a serializa accesul la obiecte de tip array. Secvenţa de cod de mai jos este un exemplu de cum se poate serializa accesul concurenţial la elementele unui şir:

```
/** Incrementeaza elementul de pe pozitia poz.*/
void increment(byte[] sir, int poz)
{
  synchronized( sir )
  {
    sir[ poz ]++ ;
  }
}
```

În exemplul anterior, maşina virtuală Java va atribui monitorul lui `sir`[4] firului de execuţie curent, înainte de a încerca să incrementeze elementul de pe poziţia `poz`. Alte fire de execuţie care ar încerca să obţină monitorul acestui şir

[3] Acest exemplu nu este concludent pentru multe bănci din România, care au nevoie de un interval de 2 zile pentru ca suma transferată dintr-un cont să apară în celălalt cont...

[4] Deşi nu sunt obiecte în adevăratul sens al cuvântului, şirurile din Java au anumite proprietăţi specifice obiectelor, printre care şi monitorul.

Digitized by Google

vor fi forţate să aştepte până când se încheie operaţia de incrementare. Trebuie însă să avem în vedere faptul că accesul la elementele şirului va fi totuşi permis în cadrul altor instrucţiuni care nu sunt sincronizate pe monitorul şirului.

Instrucţiunea `synchronized` este utilă şi atunci când dorim să modificăm un obiect fără a folosi metode sincronizate. Această situaţie poate să apară atunci când se modifică atributele publice ale unui obiect (deşi am arătat deja că utilizarea de atribute publice nu este de dorit) sau se apelează o metodă care nu este sincronizată (dar ar trebui să fie) şi noi nu o putem modifica (de exemplu, pentru că este moştenită de la un alt obiect). Iată un exemplu:

```
void metoda( OClasa obiect )
{
  synchronized( obiect )
  {
    obiect.metoda_care_ar_trebui_sa_fie_sincronizata_dar_nu_e();
  }
}
```

O metodă sincronizată este echivalentă cu o metodă nesincronizată ale cărei instrucţiuni sunt cuprinse într-un bloc `synchronized` pe monitorul obiectului (`this`). Astfel, metoda

```
synchronized void f()
{
 instructiune;
}
```

este echivalentă cu metoda

```
void f()
{
  synchronized( this )
  {
    instructiune;
  }
}
```

Observaţie: Instrucţiunea `synchronized` ne oferă posibilitatea de a utiliza monitoarele oricărui obiect Java. Totuşi, folosirea instrucţiunii `synchronized` în situaţii în care ar fi fost suficientă declararea unei metode sincronizate duce uneori la un cod greoi şi dificil de depanat. Adăugarea modificatorului `synchronized` la antetul unei metode defineşte foarte clar ce se petrece atunci când metoda este apelată.

Digitized by Google

8.3.4 Competiţie asupra monitoarelor

Se poate întâmpla ca două sau mai multe fire de execuţie să fie blocate în aşteptarea obţinerii aceluiaşi monitor. Această situaţie apare atunci când un fir de execuţie deţine monitorul unui anumit obiect. Dacă un alt fir de execuţie încearcă să execute una din metodele sincronizate ale aceluiaşi obiect, firul va fi suspendat, în aşteptarea eliberării monitorului. Dacă un al treilea fir de execuţie încearcă şi el să apeleze o metodă sincronizată a obiectului, va fi şi el suspendat. În momentul în care monitorul va deveni disponibil, vor fi două fire de execuţie care aşteaptă să îl obţină.

În situaţia în care două sau mai multe fire de execuţie aşteaptă obţinerea aceluiaşi monitor, maşina virtuală Java va trebui să aleagă doar un singur fir şi să îi atribuie acestuia monitorul. Nu există nici o garanţie în privinţa regulii după care maşina virtuală va decide care fir va fi ales. Specificaţia limbajului Java prevede ca doar un singur fir de execuţie să obţină monitorul, fără a preciza un algoritm după care maşina virtuală să aleagă firul de execuţie. De exemplu, maşina virtuală Java care rulează pe Solaris, alege firul de execuţie în funcţie de prioritate cu primul venit-primul servit în cazul în care priorităţile sunt egale. Totuşi, maşina virtuală Java de pe Win32 foloseşte algoritmii de planificare specifici Win32.

Astfel, nu este posibil să prevedem ordinea în care firele de execuţie vor obţine un monitor în cazul în care mai multe fire de execuţie sunt în aşteptarea lui, şi de aceea trebuie să evitaţi scrierea de cod care ar depinde de această ordine.

8.3.5 Sincronizarea metodelor statice

Am văzut deja că, pentru a putea fi executate, metodele care sunt declarate a fi sincronizate, sunt obligate să deţină monitorul obiectului pentru care au fost apelate. Ce putem însă spune atunci despre metodele statice, care, după cum ştim, nu trebuie neapărat apelate în conjuncţie cu un obiect (metodele statice sunt specifice claselor, şi nu obiectelor)?

Soluţia oferită de Java este simplă: în momentul în care se apelează o metodă statică sincronizată, aceasta va trebui să obţină un monitor special, care este asociat fiecărei clase în parte. Cu alte cuvinte, fiecare clasă Java are asociat un monitor care controlează accesul la metodele statice sincronizate ale clasei. Rezultă de aici că doar o singură metodă statică sincronizată se poate executa la un moment dat în cadrul unei clase.

Un fapt care merită remarcat este că, în implementarea actuală, maşina virtuală Java foloseşte pentru sincronizarea metodelor statice monitorul instanţei

Digitized by Google

de tip java.lang.Class asociat clasei respective. Aşa cum am văzut şi în paragraful 5.8.2, pagina 165, instanţele clasei java.lang.Class descriu toate clasele şi interfeţele existente în cadrul unei aplicaţii Java aflată în execuţie. Fiecare clasă sau interfaţă Java (chiar şi tipurile primitive) are asociată o instanţă de tip java.lang.Class. Obiectele de tip Class sunt construite automat de către maşina virtuală Java pe măsură ce clasele sunt încărcate în memorie. Prin intermediul unei instanţe java.lang.Class asociată unei clase se pot obţine informaţii despre clasa respectivă, cum ar fi numărul de atribute şi tipul lor, metodele clasei, pachetul din care clasa face parte etc. Instanţa java.lang.Class asociată unei clase poate fi obţinută folosind metoda getClass() definită chiar în clasa Object. Ajunşi în acest punct, vă întrebaţi, probabil, la ce ne foloseşte să avem o clasă care să descrie conţinutul fiecărei clase? Răspunsul este că această clasă este piatra de temelie a aşa-numitului Reflection API (vezi capitolul 5.8.2), care este folosit în *debugger*-e[5], serializare şi deserializare de obiecte etc.

Pentru a ne convinge de faptul că metodele statice folosesc monitorul asociat instanţei java.lang.Class, să studiem clasa StaticMonitorTest de mai jos:

```
public class StaticMonitorTest extends Thread
{
  public void run()
  {
    synchronized( getClass() )
    {
      System.out.println("Se executa run()");
      try { sleep(5000); } catch (InterruptedException _) { }
    }
  }

  public static synchronized void staticF()
  {
    System.out.println("Se executa staticF()");
    try { sleep(5000); } catch (InterruptedException _) { }
  }

  public static void main(String[] args)
  {
    new StaticMonitorTest().start();
    staticF();
  }
}
```

La rularea acestei clase pe Linux sau Win32, se afişează mesajul "Se ex-

[5] aplicaţii folosite în depanarea programelor

Digitized by Google

`ecuta staticF()`", după care urmează o pauză de 5 secunde, se afişează "`Se executa run()`" şi programul se încheie după o pauză de încă 5 secunde. Rezultă clar de aici că monitorul folosit pentru metoda statică este acelaşi cu monitorul obiectului de tip `Class` asociat. Nu se ştie dacă ne putem baza pe acest comportament şi în versiunile viitoare de Java. Un lucru este totuşi cert: două metode statice definite în cadrul aceleiaşi clase vor utiliza acelaşi monitor.

8.3.6 Monitoare şi atribute publice

Am arătat deja că declararea de atribute publice nu este de dorit. Acest lucru devine şi mai evident dacă privim problema din perspectiva accesului concurent. Atributele publice ale unui obiect pot fi accesate de către un fir de execuţie fără a beneficia de protecţia oferită de metodele sincronizate. De fiecare dată când declarăm un atribut ca fiind public, lăsăm controlul asupra actualizării acelui atribut în seama oricărui programator care ne utilizează clasa, ceea ce poate conduce cu uşurinţă la inconsistenţă la concurenţă.

Observaţie: Programatorii Java obişnuiesc adeseori să declare constante simbolice sub forma unor atribute statice de tip `final public`. Atributele declarate în acest fel nu pun probleme la concurenţă, deoarece inconsistenţele pot apărea doar în conjuncţie cu obiecte a căror valoare nu este constantă.

8.3.7 Când NU trebuie să folosim sincronizarea

Ajunşi în acest punct cu lectura acestui capitol ar trebui să fiţi deja capabili să scrieţi cod *thread-safe*[6], folosind cuvântul cheie `synchronized`. Să vedem acum când este într-adevăr necesar să folosim metode sincronizate şi care sunt dezavantajele folosirii unor astfel de metode.

Cel mai adesea, programatorii nu folosesc instrucţiunea `synchronized` deoarece scriu cod single-threaded (pe un singur fir de execuţie). De exemplu, rutinele care solicită procesorul intensiv nu beneficiază prea mult de fire multiple de execuţie. Un compilator nu are performanţe semnificativ mai bune dacă rulează pe mai multe fire de execuţie. Ca dovadă, compilatorul Java de la Sun nu conţine prea multe metode sincronizate, deoarece se consideră că rulează în special în propriul său fir de execuţie, fără a fi necesar să partajeze resursele cu alte fire de execuţie.

Un alt motiv comun pentru evitarea metodelor sincronizate este că ele nu sunt la fel de eficiente ca metodele nesincronizate. Anumite teste simple au pus

[6]cod care nu crează probleme prin execuţia pe mai multe fire

Digitized by Google

în evidenţă faptul că metodele sincronizate sunt de aproximativ patru ori mai lente decât variantele lor nesincronizate. Aceasta nu implică neapărat faptul că întreaga aplicaţie va fi de patru ori mai lentă, dar este o problemă care trebuie avută în vedere. Există programe care necesită drămuirea fiecărui gram de utilizare din sistemul pe care rulează (de exemplu, programele Java care rulează pe PDA[7]-uri). În asemenea situaţii este adeseori adecvat să eliminăm surplusul de calcule generat de metodele sincronizate acolo unde este posibil.

Îmbunătăţiri ale performanţei pot să apară şi dacă, în loc să declarăm o metodă întreagă ca fiind sincronizată, folosim un bloc `synchronized` doar pentru cele câteva instrucţiuni care trebuie să fie sincronizate. Aceasta se aplică mai ales atunci când avem metode lungi, care realizează multe calcule, şi doar o mică porţiune din ele este susceptibilă de inconsistenţă la concurenţă.

8.3.8 Blocare circulară (deadlock)

Blocarea circulară, numită uneori şi *îmbrăţişare mortală* este unul dintre cele mai grave evenimente care pot să apară într-un mediu multithreaded. Programele Java nu sunt imune la blocări circulare (deadlocks), iar programatorii trebuie să fie atenţi pentru a le evita.

O blocare circulară este o situaţie în care două sau mai multe fire de execuţie sunt blocate în aşteptarea unei resurse şi nu pot continua execuţia. În cazul cel mai simplu, avem două fire de execuţie fiecare dintre ele aşteptând eliberarea unui monitor deţinut de celălalt fir. Fiecare fir este pus în aşteptare, până când monitorul dorit devine disponibil. Totuşi, acest lucru nu se va realiza niciodată. Primul fir aşteaptă monitorul deţinut de al doilea fir, iar al doilea aşteaptă monitorul deţinut de primul. Deoarece fiecare fir este în aşteptare, nici unul nu va elibera monitorul necesar celuilalt fir.

Programul din **Listing 8.6** este un exemplu simplu menit să vă dea o idee despre cum poate să apară o blocare circulară.

Listing 8.6: Exemplu de blocare circulară

```
/** Clasa care provoaca o blocare circulara prin
* pornirea a doua fire de executie care se asteapta reciproc. */
public class Deadlock implements Runnable
{
  private Deadlock otherThread ;
```

[7]PDA = Personal Digital Assistant, cunoscute şi sub numele de Handheld PC sau Palm, reprezintă unităţi mobile de dimensiune mică având capacitatea de a stoca şi retrage informaţie. Au fost folosite în special ca nişte agende electronice, capabile să reţină adrese, întâlniri etc. Funcţionalitatea lor s-a extins în prezent, fiind capabile de operaţii mai complexe, cum ar fi vizualizarea de documente, consultarea poştei electronice (email), calcul tabelar etc.

Digitized by Google

```
6   public static void main(String[] args)
7   {
8     Deadlock d1 = new Deadlock();
9     Deadlock d2 = new Deadlock();
10    Thread t1 = new Thread(d1);
11    Thread t2 = new Thread(d2);
12
13    d1.otherThread = d2;
14    d2.otherThread = d1;
15    t1.start();
16    t2.start();
17  }
18
19
20  public synchronized void run()
21  {
22    try { Thread.sleep(2000); } catch(InterruptedException e) { }
23    System.out.println("am intrat in metoda run");
24    otherThread.syncMethod();
25  }
26
27  public synchronized void syncMethod()
28  {
29    try { Thread.sleep(2000); } catch(InterruptedException e) { }
30    System.out.println("Am intrat in metoda sincronizata");
31  }
32 }
```

În acest exemplu, metoda `main()` lansează în liniile 15-16 două fire de execuţie, fiecare dintre ele apelând metoda sincronizată `run()` a clasei `Deadlock`. Când primul fir se trezeşte (după ce a dormit 2 secunde) la linia 23 afişează mesajul de intrare în metoda `run()`, după care încearcă să apeleze metoda `syncMethod()` a celuilalt fir de execuţie la linia 24. Evident, monitorul celuilalt obiect este deţinut de al doilea fir de execuţie, aşa că primul fir stă şi aşteaptă eliberarea monitorului. În momentul în care se trezeşte, al doilea fir încearcă să apeleze `syncMethod()` a primului fir. Cum monitorul primului fir este deja deţinut de metoda `run()` a acestuia, şi al doilea fir va intra în aşteptare. Cele două fire de execuţie vor aştepta unul după altul şi nu îşi vor continua niciodată execuţia. Acest fapt este demonstrat şi de mesajele afişate la execuţia programului:

```
Am intrat in metoda run
Am intrat in metoda run
```

după care programul se blochează fără să mai facă nimic, până când este întrerupt cu CTRL-BREAK.

Digitized by Google

Observaţie: Putem vedea exact ce se întâmplă în cadrul maşinii virtuale Java folosind următorul truc valabil pentru JDK pe Solaris şi Linux (pe Windows nu funcţionează): apăsaţi `CTRL+\` în fereastra de terminal în care rulează aplicaţia Java. Aceasta are ca efect afişarea pe ecran a stării în care se află maşina virtuală. Iată o secvenţă din textul afişat de maşina virtuală după ce am apăsat `CTRL+\` la câteva secunde după lansarea aplicaţiei `Deadlock`:

```
"Thread-1" prio=1 tid=0x80b2e00 nid=0x9b5 waiting
for monitor entry [0xbe7ff000.  .0xbe7ff8b0]
at Deadlock.syncMethod (Deadlock.java:30)
at Deadlock.run (Deadlock.java:25)
at java.lang.Thread.run(Thread.java:484)
"Thread-0" prio=1 tid=0x80b2600 nid=0x9b4 waiting
for monitor entry [0xbe9ff000.  .0xbe9ff8b0]
at Deadlock.syncMethod (Deadlock.java:30)
at Deadlock.run (Deadlock.java:25)
at java.lang.Thread.run(Thread.java:484)
```

Reiese de aici clar că ambele fire de execuţie (`Thread-1` şi `Thread-0`) aşteaptă eliberarea unui monitor în cadrul metodei `run()`.

Există numeroşi algoritmi pentru prevenirea şi detectarea blocării circulare, dar prezentarea lor depăşeşte cadrul acestui capitol (există multe manuale de baze de date şi sisteme de operare care discută pe larg algoritmii de detectare a blocajelor circulare). Din păcate, maşina virtuală Java nu face nimic pentru a detecta blocajele circulare. Totuşi specificaţiile Java nu împiedică acest lucru, aşa că nu este exclus ca această facilitate să fie adăugată maşinii virtuale în versiunile viitoare.

8.4 Coordonarea firelor de execuţie

În paragraful precedent am văzut modul în care monitorul unui obiect poate fi utilizat pentru a serializa accesul la o anumită secvenţă de cod. Totuşi, monitoarele sunt mai mult decât nişte obiecte de blocare (lock), deoarece ele pot fi utilizate şi pentru a coordona execuţia mai multor fire de execuţie folosind metodele `wait()` şi `notify()`.

8.4.1 De ce este necesar să coordonăm firele de execuţie?

Firele de execuţie din cadrul unui program Java sunt adeseori interdependente; un fir de execuţie poate să depindă de un alt fir de execuţie care trebuie să încheie o anumită operaţie sau să satisfacă o cerere. De exemplu, un program

Digitized by Google

de calcul tabelar poate să realizeze un calcul care solicită intensiv procesorul pe un fir de execuţie separat. Dacă un fir de execuţie ataşat interfeţei utilizator încearcă să actualizeze imaginea afişată de program, acesta va trebui să se coordoneze cu firul care realizează calculul, începând actualizarea imaginii doar în momentul în care firul de calcul s-a încheiat.

Există foarte multe situaţii în care este util să coordonăm două sau mai multe fire de execuţie. Lista care urmează identifică unele dintre situaţiile mai uzuale:

- Zonele tampon (buffere) partajate sunt adeseori utilizate pentru a transmite date între firele de execuţie. În această situaţie, există de obicei un fir de execuţie care scrie în buffer (numit *writer* sau *producător*) şi un fir care citeşte din buffer (numit *reader* sau *consumator*). Când firul consumator citeşte din buffer, trebuie să se coordoneze cu firul producător, citind date din buffer doar *după* ce acestea au fost plasate acolo de producător. Dacă bufferul este gol, firul consumator trebuie să aştepte noi date (fără să verifice încontinuu dacă au venit date!). Firul producător trebuie să-l notifice pe consumator când a scris ceva în buffer, pentru ca acesta să poată să continue citirea;

- Dacă o aplicaţie trebuie să răspundă cu promptitudine la acţiunile utilizatorului, dar ocazional trebuie să realizeze anumite calcule numerice intensive, putem rula firul de calcul numeric cu o prioritate mică (folosind metoda `setPriority()` din clasa `Thread`). Toate firele de execuţie de prioritate mai mare care au nevoie de rezultatul calculului vor aştepta ca firul cu prioritate mai mică să se încheie, moment în care acesta va notifica toate firele de execuţie interesate că şi-a încheiat calculul;

- Un fir de execuţie poate fi construit în aşa manieră încât execută anumite operaţii ca răspuns la anumite evenimente generate de alte fire de execuţie. Dacă nu există nici un eveniment netratat, firul este suspendat (un fir de execuţie care nu face nimic nu ar trebui să consume procesor). Firele de execuţie care generează evenimente trebuie să aibă la dispoziţie un mecanism prin care să notifice firul care aşteaptă de faptul că s-a produs un eveniment.

Nu este întâmplător faptul că exemplele de mai sus au folosit în mod repetat cuvintele "aşteaptă" şi "notifică". Aceste două cuvinte exprimă conceptele care stau la baza coordonării firelor de execuţie: un fir aşteaptă producerea unui eveniment, iar un alt fir notifică producerea acelui eveniment. Aceste cuvinte, wait (aşteaptă) şi notify (notifică) sunt folosite şi în Java, ca numele metodelor care se apelează pentru a coordona firele de execuţie (`wait()` şi `notify()` din clasa `Object`).

Digitized by Google

Aşa cum am arătat în paragraful 8.3.1 din cadrul acestui capitol, fiecare obiect Java are asociat un monitor. Acest fapt devine foarte util în acest moment, deoarece vom vedea că monitoarele sunt folosite şi pentru a implementa primitivele de coordonare a firelor de execuţie. Deşi monitoarele nu sunt accesibile în mod direct programatorului, clasa `Object` furnizează două metode prin intermediul cărora putem interacţiona cu monitorul unui obiect. Aceste două metode sunt `wait()` şi `notify()`.

8.4.2 `wait()` şi `notify()`

Firele de execuţie sunt de obicei coordonate folosind conceptul de *condiţie* sau *variabilă condiţională*. O condiţie reprezintă o stare sau un eveniment fără de care firul de execuţie nu poate continua execuţia. În Java, acest model este exprimat în general sub forma

```
while( ! conditia_dupa_care_astept )
{
  wait();
}
```

La început, se verifică dacă variabila condiţională este adevărată. Dacă da, nu este necesar să se aştepte. În cazul în care condiţia nu este încă adevărată, se apelează metoda `wait()`. Când aşteptarea (`wait()`) se încheie (fie pentru că firul a fost notificat, fie pentru că a expirat timpul de aşteptare) se verifică din nou condiţia pentru a ne asigura că a devenit adevărată (dacă suntem siguri că a devenit adevărată, putem înlocui ciclul `while` cu un simplu `if`).

Apelul metodei `wait()` pe un obiect opreşte temporar firul de execuţie curent până când un alt fir apelează `notify()`, pentu a informa firul care aşteaptă că s-a produs un eveniment care ar putea schimba condiţia de aşteptare. Cât timp un fir de execuţie aşteaptă în cadrul unei instrucţiuni `wait()`, acesta este considerat de către maşina virtuală ca fiind suspendat şi nu i se alocă timp de execuţie pe procesor până în momentul în care este trezit printr-un apel al lui `notify()` dintr-un alt fir de execuţie. Apelul lui `notify()` trebuie făcut dintr-un alt fir, deoarece firul curent este suspendat, deci nu are cum să apeleze `notify()`. Apelul lui `notify()` va informa un singur fir de execuţie aflat în aşteptare de faptul că starea obiectului s-a modificat, încheind astfel aşteptarea din cadrul lui `wait()`.

Metoda `wait()` are şi două variaţiuni uşor diferite. Prima versiune acceptă un parametru de tip `long`, interpretat ca perioadă de expirare a aşteptării (timeout) măsurată în milisecunde. A doua variantă acceptă doi parametri, interpretaţi tot ca o perioadă de expirare (în milisecunde şi nanosecunde) a aşteptării. Aceste variante sunt folosite atunci când nu dorim să aşteptăm la infinit

Digitized by Google

producerea unui anumit eveniment. Dacă dorim să renunţăm la aşteptare după o anumită perioadă, va trebui să folosim una dintre metodele:

```
wait( long miliseconds )
wait( long miliseconds , int nanoseconds )
```

Un mic inconvenient este dat de faptul că aceste metode nu oferă o modalitate prin care să determinăm dacă apelul lui `wait()` s-a încheiat printr-un apel de `notify()` sau prin expirare (timeout). Totuşi această problemă poate fi uşor rezolvată, deoarece putem verifica din nou condiţia de aşteptare şi timpul curent pentru a determina care dintre cele două evenimente s-a produs.

Metodele `wait()` şi `notify()` pot fi apelate doar dacă firul de execuţie curent deţine monitorul obiectului pe care sunt apelate. Vom detalia acest aspect în paragrafele următoare.

8.4.3 Exemplu de coordonare a firelor de execuţie: problema consumator-producător

Exemplul clasic de coordonare a firelor de execuţie, folosit în majoritatea lucărilor de programare este problema bufferului de capacitate limitată. În varianta sa cea mai simplă, această problemă presupune că se utilizează un buffer (zonă tampon de memorie) pentru a comunica între două fire de execuţie sau procese (în multe sisteme de operare, bufferele de comunicare între procese au dimensiune fixă). Există un fir de execuţie care scrie date în buffer şi un altul care preia datele din buffer spre a fi prelucrate mai departe. Pentru ca bufferul să funcţioneze corect, trebuie ca firul care scrie şi cel care citeşte să fie coordonate, astfel încât:

- firul care scrie poate să adauge date în buffer până când acesta se umple, moment în care firul este suspendat;
- după ce firul care citeşte extrage date din bufferul plin, notifică firul care scrie asupra stării modificate a bufferului, care este activat şi i se permite să reia scrierea;
- firul care citeşte poate să extragă date din buffer până când acesta se goleşte, moment în care firul este suspendat;
- când firul care scrie adaugă date la bufferul gol, firul care citeşte este notificat asupra stării modificate a bufferului, este activat şi i se permite să reia citirea.

Digitized by Google

Vom prezenta în continuare trei clase Java care rezolvă problema bufferului de capacitate limitată. Acestea sunt denumite în mod clasic `Producer` (clasa care implementează firul de execuţie care scrie în buffer), `Consumer` (clasa care implementează firul de execuţie care citeşte din buffer) şi `Buffer` (clasa centrală, care oferă metode pentru a adăuga şi citi date în şi din buffer).

Să începem cu clasa `Producer`, care este extrem de simplă:

```
public class Producer implements Runnable
{
  private Buffer buffer;

  public Producer(Buffer buffer)
  {
    this.buffer = buffer ;
  }

  public void run()
  {
    for (int i=0; i<250; i++)
    {
      buffer.put((char)('A' + (i%26)));
    }
  }
}
```

Clasa `Producer` implementează interfaţa `Runnable`, ceea ce reprezintă un indiciu al faptului că va rula pe un fir de execuţie separat. În momentul în care se apelează metoda `run()` a clasei `Producer`, se va scrie rapid în `Buffer` o succesiune de 250 de caractere. Dacă clasa `Buffer` nu este capabilă să reţină toate cele 250 de caractere, rămâne în responsabilitatea metodei `put()` a bufferului să realizeze coordonarea adecvată a firelor de execuţie (vom vedea imediat cum).

Clasa `Consumer` este la fel de simplă ca şi clasa `Producer`:

```
public class Consumer implements Runnable
{
  private Buffer buffer;

  public Consumer(Buffer b)
  {
    buffer = b;
  }

  public void run()
  {
    for (int i=0; i<250; i++)
    {
      System.out.println(buffer.get());
```


Digitized by Google

```
    }
  }
}
```

Ca şi în cazul clasei `Producer`, clasa `Consumer` implementează de asemenea interfaţa `Runnable`. Metoda ei `run()` citeşte cu lăcomie 250 de caractere din `Buffer`. În situaţia în care `Consumer` încearcă să preia caractere dintr-un `Buffer` gol, metoda `get()` a clasei `Buffer` este responsabilă de a coordona firul de execuţie al consumatorului.

Iată acum care este codul pentru clasa `Buffer`. Două dintre metodele ei, `get()` şi `put()`, au fost deja amintite.

Listing 8.7: Clasa `Buffer`

```
public class Buffer
{
  private char[] buf ;//retine elementele din buffer
  private int last  ; //ultima pozitie ocupata

  /** Construieste un buffer cu size elemente */
  public Buffer(int size)
  {
    buf = new char[size];
    last = 0;
  }

  /** Intoarce true daca bufferul e plin.*/
  public boolean isFull ()
  {
    return (last == buf.length);
  }

  /** Intoarce true daca bufferul e gol. */
  public boolean isEmpty()
  {
    return (last == 0);
  }

  /** Adauga caracterul c la buffer. Daca bufferul
    * este plin, atunci se asteapta pana cand se obtine
    * o notificare de la metoda get(). */
  public synchronized void put(char c)
  {
    while(isFull())
    {
      try
      {
              wait();
      }
```


Digitized by Google

```
      catch(InterruptedException e) { }
    }
    buf[last++] = c;
    // anunta firul consumator ca bufferul nu mai e gol
    notify();
  }

  /** Extrage primul caracter din Buffer. Daca bufferul
    * este gol, se asteapta pana cand se obtine o notificare
    * de la metoda get(). */
  public synchronized char get()
  {
    while(isEmpty())
    {
      try
      {
        wait();
      }
      catch(InterruptedException e) { }
    }
    char c =buf[0];
    //deplaseaza elementele din buffer la stanga
    System.arraycopy(buf, 1, buf, 0, --last);
    //anunta firul producator ca bufferul nu mai e plin
    notify();
    return c;
  }
}
```

Observaţie: Cititorii atenţi, aflaţi la prima confruntare cu metodele `wait()` şi `notify()` ar putea observa o contradicţie. S-a menţionat deja faptul că pentru a putea apela metodele `wait()` şi `notify()`, firul de execuţie trebuie să deţină monitorul obiectului. Astfel, dacă un fir de execuţie obţine monitorul unui obiect şi intră apoi în aşteptare printr-un apel al metodei `wait()`, cum va putea alt fir de execuţie să obţină monitorul obiectului pentru a putea notifica primul fir? Oare monitorul nu este încă în posesia firului care aşteaptă, împiedicând astfel al doilea fir în a-l obţine? Răspunsul la acest aparent paradox este dat de modul în care este implementată în Java metoda `wait()`: aceasta eliberează temporar monitorul după ce a fost apelată, şi obţine din nou monitorul înainte de a se încheia. Astfel, metoda `wait()` permite şi altor fire de execuţie să obţină monitorul obiectului şi (eventual) să apeleze metoda `notify()`.

Clasa `Buffer` este doar o simplă zonă tampon de memorie şi nimic mai mult. Se pot adăuga elemente în buffer, folosind metoda `put()` şi se pot extrage elemente din buffer folosind metoda `get()`.

Digitized by Google

Observaţi modul în care s-au utilizat `wait()` şi `notify()` în aceste metode. În cadrul metodei `put()`, atâta timp cât bufferul este plin, se intră în aşteptare. Astfel, nu se vor mai putea adăuga noi elemente în buffer până în momentul în care firul producător nu este notificat (la linia 60) de către consumator că au fost extrase elemente din buffer. Apelul metodei `notify()` din finalul metodei `get()` are rolul de a activa firul de execuţie care aşteaptă în cadrul metodei `put()` la linia 34 (dacă acesta există), permiţându-i astfel să adauge un nou caracter în `Buffer`. Un raţionament absolut similar este valabil şi pentru metoda `get()`, în care se aşteaptă cât timp bufferul este gol, după care se extrage un caracter şi se notifică noua stare a bufferului.

Clasa `BufferTest` de mai jos, construieşte un `Buffer`, după care porneşte pe fire de execuţie separate producătorul şi consumatorul:

```
public class TestBuffer
{
  public static void main(String args[])
  {
    Buffer b = new Buffer( 50 ) ;
    new Thread( new Producer( b ) ).start() ;
    new Thread( new Consumer( b ) ).start() ;
  }
}
```

Rezultatul afisat la consolă (în cadrul clasei `Consumer`) confirmă faptul că firele de execuţie au fost corect coordonate:

```
  A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z A
B C D E F G H I J K L M N O P Q R S T U V W X Y Z A B
C D E F G H I J K L M N O P Q R S T U V W X Y Z A B C
D E F G H I J K L M N O P Q R S T U V W X Y Z A B C D
E F G H I J K L M N O P Q R S T U V W X Y Z A B C D E
F G H I J K L M N O P Q R S T U V W X Y Z A B C D E F
G H I J K L M N O P Q R S T U V W X Y Z A B C D E F G
H I J K L M N O P Q R S T U V W X Y Z A B C D E F
```

Implementarea metodei `get()` din clasa `Buffer` nu este foarte eficientă. La fiecare apel al acestei metode, toate celelalte caractere din buffer trebuie deplasate către stânga. O problemă de la finalul acestui capitol propune o implementare mai eficientă a clasei `Buffer` folosind o coadă alocată static (detalii despre cozi în cel de-al doilea volum, dedicat structurilor de date şi algoritmilor).

Digitized by Google

8.4.4 Deţinerea monitoarelor

Există o restricţie importantă în folosirea metodelor `wait()` şi `notify()`: aceste metode pot fi apelate doar dacă firul de execuţie curent deţine monitorul obiectului. În multe situaţii, metodele `wait()` şi `notify()` sunt apelate în cadrul unor metode sincronizate (deţinerea monitorului fiind astfel asigurată), ca în exemplul de mai jos:

```
public synchronized void method()
{
  ...
  while (!condition)
  {
    wait ();
  }
  ...
}
```

În situaţia de mai sus, modificatorul `synchronized` ne asigură de faptul că firul de execuţie care apelează `wait()` deţine deja monitorul în momentul apelului.

Ce se întâmplă însă dacă se încearcă apelul `wait()` sau `notify()` fără a deţine monitorul obiectului (de exemplu, în cadrul unei metode nesincronizate)? Compilatorul nu poate să îşi dea seama dacă există un apel ilegal (fără a deţine monitorul) al acestor metode (de exemplu se poate ca o metodă nesincronizată să apeleze `wait()`, dar această metodă să fie apelată de o metodă sincronizată). Aşadar, singura posibilitate este ca, în timpul execuţiei, dacă se întâlneşte un apel al metodelor `wait()` sau `notify()` fără a deţine monitorul obiectului să se genereze o excepţie. Această excepţie se numeşte `IllegalMonitorStateException` şi este aruncată de maşina virtuală Java ori de câte ori întâlneşte un apel ilegal al metodelor `wait()` sau `notify()`. Clasa `MonitorTest` de mai jos evidenţiază ce se petrece în momentul în care se apelează metoda `wait()` fără a fi deţinut în prealabil monitorul clasei:

```
public class MonitorTest
{
  public static void main(String[] args)
  {
    MonitorTest mt = new MonitorTest();
    mt.method();
  }

  public void method()
  {
    try
    {
      wait();
```


Digitized by Google

```
    }
    catch(InterruptedException e)
    {
    }
  }
}
```

Dacă veţi încerca să executaţi această clasă, veţi primi următorul mesaj:

```
Exception in thread "main"
    java.lang.IllegalMonitorStateException
    at java.lang.Object.wait(Native Method)
    at java.lang.Object.wait(Object.java:420)
    at MonitorTest.method(MonitorTest.java:13)
    at MonitorTest.main(MonitorTest.java:6)
```

Pentru a evita o astfel de excepţie, este necesar ca de fiecare dată când apelaţi metoda `wait()` pe un obiect să vă asiguraţi că firul de execuţie deţine monitorul acelui obiect.

8.4.5 Utilizarea instrucţiunii `synchronized`

Instrucţiunea `synchronized` poate fi şi ea folosită pentru a realiza coordonarea firelor de execuţie. În cele mai multe cazuri utilizarea metodelor sincronizate este suficientă, şi datorită faptului că sintaxa lor este mai simplă ele sunt de preferat instrucţiunii `synchronized`. Există totuşi două situaţii în care utilizarea metodelor sincronizate nu este de dorit, sau nu poate rezolva problema.

Dacă metoda care conţine un `wait()` sau `notify()` este foarte lungă (şi necesită un timp de execuţie semnificativ) este preferabil să se sincronizeze doar secvenţa de cod care este strict necesară pentru a menţine consistenţa la concurenţă.

Adeseori este necesar să coordonăm metode din obiecte diferite. În această situaţie, utilizarea de metode sincronizate nu are nici un sens, deoarece apelul lui `notify()` din cadrul unei metode sincronizate a unui obiect nu are nici un efect asupra firului de execuţie aflat în aşteptare în cadrul unui `wait()` pe un *alt* obiect (metodele obţin monitoare distincte)!

Să presupunem de exemplu că avem la dispoziţie o clasă simplă `Buffer1` identică din punct de vedere al interfeţei cu clasa `Buffer` din paragraful 8.4.3, dar care a fost concepută şi scrisă pentru a funcţiona corect pe un singur fir de execuţie:

Digitized by Google

```
public class Buffer1
{
  private char[] buf; //retine elementele din buffer
  private int last; //ultima pozitie ocupata

  /** Construieste un buffer cu size elemente */
  public Buffer1(int size)
  {
    buf = new char[size];
    last = 0;
  }

  /** Intoarce true daca bufferul e plin.*/
  public boolean isFull()
  {
    return (last == buf.length);
  }

  /** Intoarce true daca bufferul e gol. */
  public boolean isEmpty()
  {
    return (last == 0);
  }

  /** Adauga caracterul c la buffer. */
  public synchronized void put(char c)
  {
    buf[last++] = c;
  }

  /** Extrage primul caracter din Buffer. */
  public synchronized char get()
  {
    char c = buf[0];
    //deplaseaza elementele din buffer la stanga
    System.arraycopy(buf, 1, buf, 0, --last);
    return c;
  }
}
```

Observaţi că, spre deosebire de clasa `Buffer` din paragraful 8.4.3, această clasă nu conţine nici o linie de cod pentru a coordona în vreun fel producătorul şi consumatorul. Rămâne astfel în sarcina producătorului şi a consumatorului să îşi coordoneze accesul la buffer. Având în vedere că producătorul şi consumatorul sunt implementate de clase diferite, utilizarea metodelor sincronizate în vederea coordonării nu este posibilă. Coordonarea ar trebui să se facă folosind `wait()` şi `notify()` pe acelaşi obiect. Care obiect? Chiar instanţa clasei `Buffer1`. Codul de mai jos descrie clasa `Producer1`, care este o variantă

Digitized by Google

modificată a clasei `Producer`:

```
1 public class Producer1 implements Runnable
2 {
3   private Buffer1 buffer;
4
5   public Producer1(Buffer1 buffer)
6   {
7     this.buffer = buffer;
8   }
9
10  public void run()
11  {
12    for (int i=0; i<250; i++)
13    {
14      synchronized( buffer )
15      {
16        while ( buffer.isFull() )
17        {
18          try
19          {
20            buffer.wait();
21          }
22          catch( InterruptedException e )
23          {}
24        }
25        buffer.put((char)('A' + (i%26)));
26        buffer.notify();
27      }
28    }
29  }
30 }
```

Diferenţa faţă de clasa `Producer` apare la metoda `run()`, care conţine cod suplimentar pentru a realiza sincronizarea cu consumatorul. În primul rând, la linia 14 se solicită monitorul bufferului, folosind instrucţiunea

```
synchronized(buffer)
{
  ...
}
```

Instrucţiunea `synchronized` este utilizată în acest caz din două motive. În primul rând, trebuie asigurată consistenţa la concurenţă, prin evitarea situaţiei de a se scrie şi citi în buffer în acelaşi timp. În al doilea rând, instrucţiunea `synchronized` garantează obţinerea monitorului obiectului buffer, pe care se va putea apoi apela `wait()` în cazul în care bufferul este plin (linia 20), respectiv `notify()` după ce s-a citit din buffer (linia 26). Remarcaţi faptul că `wait()` şi `notify()` au fost apelate în conjuncţie cu obiectul `buffer`:

Digitized by Google

```
buffer.wait()
buffer.notify()
```

deoarece coordonarea se face utilizând monitorul bufferului.

Implementarea consumatorului este similară cu cea a producătorului şi este realizată de clasa `Consumer1` de mai jos:

```
public class Consumer1 implements Runnable
{
  private Buffer1 buffer;

  public Consumer1(Buffer1 buffer)
  {
    this.buffer = buffer ;
  }

  public void run()
  {
    for (int i=0; i<250; i++)
    {
      synchronized( buffer )
      {
        while ( buffer.isEmpty() )
        {
          try
          {
            buffer.wait() ;
          }
          catch( InterruptedException e )
          {}
        }
        System.out.print(buffer.get() + "-");
        buffer.notify() ;
      }
    }
  }
}
```

Pentru a testa noua rezolvare a problemei bufferului, am creat clasa `TestBuffer1`, absolut similară clasei `TestBuffer`:

```
public class TestBuffer1
{
  public static void main(String args[])
  {
    Buffer1 b = new Buffer1( 2 ) ;
    new Thread( new Producer1( b ) ).start() ;
    new Thread( new Consumer1( b ) ).start() ;
  }
}
```


Digitized by Google

La execuţia acestei clase, se va afişa aceeaşi secvenţă de caractere ca şi la rezolvarea precedentă, ceea ce dovedeşte că şi această abordare este viabilă.

8.4.6 Un alt exemplu: consumatori şi producători multipli

Este posibil ca mai multe fire de execuţie să fie în aşteptare (`wait()`) pe acelaşi obiect. Această situaţie poate să apară dacă mai multe fire de execuţie aşteaptă apariţia aceluiaşi eveniment, sau dacă există mai multe fire de execuţie care partajează o resursă unică. Să luăm din nou exemplul clasei `Buffer` prezentată în secţiunea 8.4.3. Bufferul era utilizat de către un singur producător şi un singur consumator. Ce s-ar petrece însă, dacă ar exista mai mulţi producători? Dacă bufferul se umple, ar putea exista mai mulţi producători care să încerce să adauge date în `Buffer`. Toţi aceştia ar sta blocaţi în cadrul metodei `put()`, aşteptând ca un consumator să elibereze ceva spaţiu în `Buffer`.

În momentul în care se apelează `notify()` pe un obiect, pot exista zero, unul sau mai multe fire de execuţie blocate în cadrul unui `wait()` pe acel obiect. Dacă nu există nici un fir de execuţie în aşteptare, apelul metodei `notify()` nu are nici un efect. Dacă un singur fir de execuţie este în aşteptare, acesta va fi notificat şi va începe să aştepte eliberarea monitorului de către firul de execuţie care a apelat `notify()`. Dacă există două sau mai multe fire de execuţie în aşteptare, maşina virtuală Java va alege un singur fir de execuţie, pe care îl va notifica.

Cum se face alegerea firului de execuţie care va fi notificat? Situaţia este aceeaşi cu cea în care mai multe fire de execuţie aşteaptă eliberarea unui monitor: comportamentul nu este specificat. Totuşi, implementările curente ale maşinii virtuale Java, folosesc un algoritm bine definit. Astfel, maşina virtuală de pe Solaris va alege firul de execuţie cu prioritatea cea mai mare şi îl va notifica. Dacă există mai mult de un fir de execuţie cu aceeaşi prioritate, va fi ales firul care a intrat primul în `wait()`. Situaţia în cazul maşinii virtuale de pe Windows este ceva mai complicată, deoarece în acest caz se folosesc algoritmii de planificare specifici sistemului de operare.

Astfel, deşi se poate prezice care dintre fire de execuţie va fi notificat, nu trebuie în nici un caz să scriem cod care să se bazeze pe astfel de deducţii. Singurul fapt pe care ne putem baza este că doar un singur fir de execuţie va fi notificat la apelul lui `notify()` (dacă există cel puţin un fir de execuţie în aşteptare).

Digitized by Google

Utilizarea metodei `notifyAll()`

Există situaţii în care vom dori notificarea fiecărui fir de execuţie care aşteaptă la un moment dat pe un obiect. Interfaţa clasei `Object` oferă o metodă care face exact acest lucru: `notifyAll()`. În timp ce metoda `notify()` trezeşte un singur fir de execuţie, metoda `notifyAll()` va trezi toate firele de execuţie aflate în acel moment în aşteptare pe monitorul obiectului.

Un exemplu concret în care este necesară utilizarea lui `notifyAll()` îl constituie problema bufferului cu producători şi consumatori multipli. În versiunea prezentată în cadrul paragrafului 8.4.3, clasa `Buffer` utiliza metoda `notify()` pentru a trimite o notificare unui singur fir de execuţie aflat în aşteptare pe un buffer plin sau gol. Totuşi, în practică, nu există nici o garanţie că doar un singur fir de execuţie se află în aşteptare, deoarece pot exista consumatori şi producători multipli. Iată o altă variantă a clasei `Buffer` denumită (din lipsă de imaginaţie) `Buffer2`, care foloseşte `notifyAll()` şi este capabilă să coordoneze consumatori şi producători multipli.

Listing 8.8: Clasa `Buffer2`

```
public class Buffer2
{
  private char[] buf; //contine elementele din buffer
  private int last = 0; // ultima pozitie ocupata
  private int writersWaiting = 0; //nr. fire ce asteapta in put()
  private int readersWaiting = 0; //nr. fire ce asteapta in get()

  /** Construieste un buffer cu size elemente */
  public Buffer2(int size)
  {
    buf = new char[size];
  }

  /** Intoarce true daca bufferul e plin.*/
  public boolean isFull()
  {
    return (last == buf.length);
  }

  /** Intoarce true daca bufferul e gol. */
  public boolean isEmpty()
  {
    return (last == 0);
  }

  /** Adauga caracterul c la buffer. Daca bufferul este plin,
    * atunci se incrementeaza <code>writersWaiting</code>
    * si se asteapta pana cand se obtine
```


Digitized by Google

```
29      * o notificare de la metoda get(). */
30    public synchronized void put(char c)
31    {
32      while( isFull() )
33      {
34        try
35        {
36          writersWaiting++;
37          wait();
38        }
39        catch( InterruptedException e )
40        { }
41        writersWaiting--;
42      }
43      buf[last++] = c;
44      if (readersWaiting > 0)
45      {
46        notifyAll();
47      }
48    }
49
50    /** Extrage primul caracter din Buffer. Daca bufferul
51      * este gol, se incrementeaza readersWaiting si se
52      * asteapta pana cand se obtine o notificare
53      * de la metoda get(). */
54    public synchronized char get()
55    {
56      while(isEmpty())
57      {
58        try
59        {
60          readersWaiting++;
61          wait();
62        }
63        catch(InterruptedException e)
64        { }
65        readersWaiting--;
66      }
67      char c = buf[0];
68      System.arraycopy(buf, 1, buf, 0, --last);
69      if (writersWaiting > 0)
70      {
71        notifyAll();
72      }
73      return c;
74    }
75  }
```

După cum aţi observat, metodele `get()` şi `put()` au fost făcute mai inteligente. Ele verifică mai întâi dacă este necesară vreo notificare (liniile 44 şi

Digitized by Google

69), după care utilizează `notifyAll()` pentru a difuza notificarea la toate firele de execuţie aflate în aşteptare.

8.4.7 Utilizarea lui `InterruptedException`

Aţi observat cu siguranţă utilizarea consecventă de-a lungul acestui capitol a clasei `InterruptedException`. Dacă veţi examina modul în care este declarată metoda `wait()` în clasa `Object`, veţi înţelege de ce:

```
public final void wait() throws InterruptedException
```

`wait()` declară că ar putea să arunce o `InterruptedException`. Aceastǎ excepţie se generează atunci când se apelează metoda `interrupt()` a clasei `Thread`. Limbajul Java oferă posibilitatea de a întrerupe aşteptarea de orice natură (*wait*, *sleep*) a unui fir de execuţie prin generarea unei excepţii de tipul `InterruptedException` într-un alt fir de execuţie, folosind metoda `interrupt()`. Această metodă este utilizată pentru a trezi firele de execuţie aflate în aşteptare în cadrului unui `wait()`, `sleep()` sau a altor operaţii (de exemplu citire de la un socket).

Rezumat

Este deosebit de important să distingem situaţiile în care este cazul să se utilizeze mai multe fire de execuţie de situaţiile în care trebuie să le evităm. Cel mai important motiv pentru utilizarea firelor de execuţie este necesitatea de a realiza mai multe operaţii, a căror rulare amalgamată va produce o utilizare mai eficientă a maşinii de calcul (incluzând aici şi posibilitatea de a distribui operaţiile pe mai multe procesoare) sau va îmbunătăţi interacţiunea cu utilizatorul.

Totuşi, utilizarea firelor de execuţie are şi dezavantaje, cum ar fi scăderea vitezei din cauza aşteptării după resurse partajate şi creşterea utilizării procesorului pentru a le gestiona.

Un alt avantaj important al firelor de execuţie este că, aşa cum am menţionat la începutul capitolului, ele înlocuiesc schimbările de context mari consumatoare de resurse generate de procese (de ordinul miilor de instrucţiuni) cu schimbări de context rapide (de ordinul sutelor de instrucţiuni). Având în vedere faptul că toate firele de execuţie din cadrul unui proces partajează aceeaşi zonă de memorie, o schimbare de context nu implică în acest caz decât modificarea punctului de execuţie şi a variabilelor locale. Pe de altă parte, o schimbare de context în cazul proceselor implică interschimbarea întregului spaţiu de memorie.

Digitized by Google

Firele de execuţie reprezintă o lume nouă, iar a învăţa să le utilizezi este echivalent cu asimilarea unui nou limbaj de programare. Una dintre principalele dificultăţi legată de firele de execuţie este dată de faptul că mai multe fire pot partaja aceleaşi resurse, cum ar fi atributele unui obiect, iar în acest caz trebuie să vă asiguraţi că nu se poate întâmpla ca mai multe fire să încerce să acceseze resursa în acelaşi timp. Aceasta presupune utilizarea atentă a cuvântului cheie `synchronized`, care este o unealtă extrem de utilă, dar care trebuie înţeleasă în profunzime, deoarece poate să provoace situaţii subtile de blocaj circular.

Capitolul de faţă este departe de a epuiza toate subiectele legate de firele de execuţie. Nu am vorbit aici despre prioritatea firelor de execuţie, despre grupuri de fire de execuţie şi clasa `ThreadGroup` sau despre fire de execuţie *daemon*. Veţi găsi informaţii despre toate aceste subiecte în excelenta carte a lui Bruce Eckel, *Thinking in Java* (vezi [Eckel]). Pentru o prezentare mai avansată a firelor de execuţie, consultaţi *Concurrent Programming in Java*, de Doug Lea, Addison-Wesley, 1997.

Noţiuni fundamentale

blocaj circular (deadlock): situaţie în care două sau mai multe fire de execuţie aşteaptă eliberarea a două sau mai multe resurse pe care le deţin reciproc (deci nu vor fi niciodată eliberate).

coordonare: termen utilizat pentru a desemna dirijarea execuţiei a două sau mai multe fire de execuţie folosind metodele `wait()` şi `notify()`.

fir de execuţie (thread) : un flux de execuţie în cadrul unui proces.

inconsistenţă la concurenţă (race condition): eroare de programare care cauzează comportamentul eronat sau imprevizibil al unui program. Mai exact, această eroare apare în cazul în care rezultatul unei operaţii depinde de factori temporali imprevizibili, cum ar fi ordinea în care s-a alocat timp pentru execuţia firelor de execuţie.

monitor: obiect special utilizat pentru a realiza excluderea reciprocă a unui grup de metode.

partajare de resurse: două sau mai multe fire de execuţie au acces la aceleaşi resurse (de exemplu, la atributele unui obiect).

proces: program de sine stătător, care dispune de propriul lui spaţiu de adrese.

serializare: constă în a impune accesul unui singur fir de execuţie la un moment dat la o resursă în acelaşi timp.

sincronizare: reprezintă coordonarea unor evenimente astfel încât doar un anumit eveniment se produce la un moment dat.

Digitized by Google

Erori frecvente

1. Cea mai frecventă eroare este că programatorul nu este conştient că rulează într-un mediu multithreaded (de exemplu în cazul unui servlet Java) şi scrie cod ignorând complet inconsistenţele la concurenţă.

2. Mulţi începători definesc firele de execuţie corect, dar uită să le pornească.

3. Din cauza grabei sau a lipsei de experienţă, adeseori nu sunt detectate secvenţele de cod care pot conduce la inconsistenţă la concurenţă.

4. Nu este suficient să sincronizăm modificatorii unui atribut pentru a obţine consistenţa la concurenţă. Trebuie sincronizate şi secvenţele de cod care accesează acel atribut.

5. Apelarea metodei `wait()` pe alt obiect decât cel pe care s-a făcut sincronizarea va genera o `IllegalMonitorStateException`.

6. Apelul lui `notify()` pe un obiect pentru a notifica un fir de execuţie care se află în aşteptare pe un alt obiect, nu va avea nici un efect.

Exerciţii

Pe scurt

1. Care este diferenţa între fire de execuţie şi procese?

2. Ce interfaţă Java trebuie implementată de către firele de execuţie?

3. Ce metodă este apelată pentru a porni un fir de execuţie?

4. Ce metodă este apelată de către maşina virtuală la pornirea unui fir de execuţie?

5. Când trebuie utilizat cuvântul cheie `synchronized`?

6. Când este de preferat instrucţiunea `synchronized()` metodelor sincronizate?

7. Ce este un monitor?

8. Care este diferenţa între coordonarea şi sincronizarea firelor de execuţie?

Digitized by Google

În practică

1. Creaţi două clase derivate din `Thread`. Metoda `run()` a primei clase obţine o referinţă către cea de-a doua clasă, după care apelează `wait()` pe acest obiect. Metoda `run()` a celei de-a doua clase va apela metoda `notifyAll()` pentru primul fir după câteva secunde, astfel încât acesta să poată afişa un mesaj.

2. Construiţi o clasă `AccountTest` care să pună în evidenţă problemele de inconsistenţă la concurenţă pe care le are clasa `Account` din paragraful 8.3.2.

3. Modul în care este implementată metoda `get()` în clasa `Buffer` din paragraful 8.4.3 este ineficient, deoarece necesită deplasarea tuturor elementelor din buffer către stânga (având astfel complexitatea $O(n)$). Modificaţi clasa `Buffer` pentru a reţine elementele sub forma unei cozi (detalii despre cozi în volumul al doilea, dedicat structurilor de date şi algoritmilor) astfel încât metodele `put()` şi `get()` să aibă complexitatea $O(1)$.

4. Clasele `Buffer1`, `Consumer1` şi `Producer1` din paragraful 8.4.5 constituie o implementare a problemei bufferului cu un singur producător şi un singur consumator în care sincronizarea şi coordonarea firelor de execuţie este făcută în cadrul producătorului şi consumatorului. Adaptaţi această soluţie pentru a funcţiona şi în cazul problemei bufferului cu producători şi consumatori multipli fără a modifica clasa `Buffer`!

   Indicaţie
   Se vor adăuga două atribute statice în cadrul claselor `Producer` şi `Consumer`, `writersWaiting` respectiv `readersWaiting`, care vor fi incrementate şi decrementate de câte ori un fir întră în aşteptare sau este trezit. Pentru a asigura consistenţa la concurenţă, trebuie să se definească metode statice sincronizate care să incrementeze respectiv să decrementeze aceste variabile. Coordonarea consumatorilor şi producătorilor se va face utilizând `wait()` şi `notifyAll()`, care vor fi apelate pe monitorul clasei `Buffer` (ca în paragraful 8.4.5).

5. Problema damelor constă în a aşeza n dame pe o tablă de şah de dimensiune $n \times n$, astfel încât acestea să nu se atace între ele. Rezolvaţi această problemă astfel încât pentru fiecare posibilitate de a aşeza o damă pe o coloană în cadrul unei linii să se pornească un fir de execuţie separat

Digitized by Google

(practic, se vor încerca în paralel toate variantele de a aşeza dama pe o anumită line)!

Proiecte de programare

1. Scrieţi o variantă generalizată a clasei `Buffer`, numită `MessageQueue`, care permite în plus comunicarea între producători şi consumatori pe anumite "canale" independente. Aceasta înseamnă că clasa `MessageQueue` va putea fi utilizată drept suport de comunicare între categorii distincte de fire de execuţie. De exemplu, un canal al clasei `MessageQueue` (să-l numim `LogChannel`) va putea fi folosit de către fire de execuţie (producători) care doresc să scrie mesaje de log. Pot exista mai mulţi consumatori care sunt înscrişi la acest canal şi preiau mesajele din coadă pentru a le prelucra într-un anumit mod. De exemplu, un consumator poate scrie mesajele de log într-o fereastră utilizator, în timp ce altul le va putea scrie într-un fişier text. Mai precis, clasa `MessageQueue` va trebui să implementeze interfaţa `IMessageQueue` de mai jos:

```java
public interface IMessageQueue
{
  /** Creaza un canal nou cu numele name.
    * Se initializeaza o coada noua de
    * asteptare pentru a stoca mesajele
    * care vin pe acest canal.
    */
  public Channel createChannel( String name );

  /** Se apeleaza pentru a inchide un canal */
  public void destroyChannel( Channel channel );

  /** Asteapta aparitia urmatorului mesaj, dupa care
    * intoarce mesajul respectiv.
    */
  public Object getMessage( Channel channel );

  /** Adauga un mesaj in coada pentru un anumit canal. */
  public void putMessage( Channel channel, Object message );
}
```


Digitized by Google

Anexe

Digitized by Google

Digitized by Google

A. Mediul Java şi uneltele ajutătoare

> Geniul nu se mulţumeşte să constate. El oferă soluţii.
>
> Robert Graves

Scopul acestei anexe este de a veni în ajutorul celor care încep să programeze în Java şi a le indica cele mai adecvate unelte, necesare pentru a începe dezvoltarea de aplicaţii Java. În cadrul acestei anexe vom prezenta:

- Cele mai populare editoare şi medii integrate Java;
- Cum se instalează Java pe Windows şi Linux;
- Cum se pot compila automat aplicaţiile Java folosind `ant;`
- *Site*-uri web care conţin informaţii utile oricărui dezvoltator Java.

A.1 Editarea codului sursă

Pentru editarea codului sursă al unui program Java este suficient un simplu editor de fişiere text, cum ar fi *Notepad* (pe Windows) sau *pico* (pe Linux). Totuşi, majoritatea programatorilor doresc facilităţi superioare celor pe care le poate oferi un astfel de editor de fişiere text. Există două variante în această situaţie: utilizarea unor editoare mai performante (paragraful A.1.1) sau utilizarea unor medii integrate de dezvoltare a aplicaţiilor Java (paragraful A.1.2).

A.1.1 Editoare

Cea mai importantă facilitate oferită de următoarele editoare este "syntax highlighting" (colorarea textului în funcţie de ceea ce reprezintă - de exemplu un

Digitized by Google

comentariu, numele unei clase, un cuvânt rezervat etc.). De asemenea, primele două editoare menţionate în tabelul de mai jos oferă facilităţi suplimentare cum ar fi compilarea şi execuţia codului, organizarea fişierelor în proiecte, interfaţă pentru `ant` (vezi paragraful A.3.1) etc. Puteţi afla mai multe despre fiecare editor în parte vizitând situl asociat, furnizat de tabelul de mai jos.

Editor	URL	OS/JDK	Observaţii
jEdit	jedit.org	J2SDK 1.3 şi 1.4	Este scris în Java şi rulează pe orice sistem care dispune de Java 2 versiunile 1.3 sau 1.4 (recomandat)
Jext	jext.org	J2SDK 1.3	Versiunea pentru Windows include un mediu Java
NEdit	nedit.org	Linux	Este scris în C şi este foarte rapid

A.1.2 Medii integrate IDE (Integrated Development Environments)

Mediile integrate IDE oferă mult mai multe facilităţi decât un simplu editor. Practic, folosind un mediu IDE, programatorului îi sunt puse la dispoziţie toate elementele de care are nevoie pentru a crea aplicaţii Java profesionale: *debugger*-e pentru executarea pas cu pas a aplicaţiilor, modalităţi de a vedea în orice moment ierarhia de clase şi metode existente în cadrul unui proiect, completarea automată a codului (code completion), compilare incrementală (codul este compilat în timp ce îl scrieţi) etc. Dintre toate aceste produse, noi recomandăm cu căldură mediul Eclipse care este uşor de utilizat şi dispune de o veritabilă armată de facilităţi deosebit de utile.

Digitized by Google

Aplicaţie	URL	OS/JDK	Observaţii
Eclipse	eclipse.org	J2SDK 1.3	Inclus şi în produsele IBM VisualAge
Forte for Java (Sun ONE)	forte.sun.com/ffj	J2SDK 1.3	Bazat pe NetBeans
JCreator	jcreator.com	Windows	Scris în C, este foarte rapid
JBuilder	borland.com/ jbuilder/personal	Windows, Linux	Varianta "Personal Edition" este disponibilă gratuit.
NetBeans IDE	netbeans.org/ide	J2SDK 1.3	
VisualAge for Java	ibm.com/software /ad/vajava	Windows	Cea mai nouă versiune este 4.0

A.2 Instalarea mediului Java

Pentru a dezvolta şi a rula programe Java, trebuie să instalaţi mediul de programare Java aflat în *kit*-ul de instalare al Java 2 SDK. Ultima versiune de Java disponibilă la momentul scrierii acestei lucrări este 1.4, iar kitul este disponibil gratuit la adresa `http://java.sun.com/j2se/1.4/download.html`. Fiecare sistem de operare are propriul lui kit de instalare, care va fi descărcat separat de la adresa anterioară. Sun oferă kit-uri de instalare pentru următoarele platforme:

- Windows;
- Linux;
- Solaris.

După ce descărcaţi kitul de instalare specific sistemului dumneavoastră de operare, puteţi începe instalarea mediului Java, ajutându-vă la nevoie de următoarele două secţiuni ce descriu instalarea Java 2 SDK v1.4.0 (cea mai recentă versiune testată) pe două sisteme de operare frecvent folosite: Windows şi Linux.

Digitized by Google

A.2.1 Instalarea sub Windows

Java 2 SDK pentru Windows poate fi utilizat sub versiunile Windows 95, Window 98 (prima şi a doua ediţie), Windows NT 4.0 cu ServicePack 5, Windows ME, Windows XP Home, Windows XP Professional, Windows 2000 Professional sau Windows 2000 Advanced Server, pe tehnologie Intel. Pentru a putea rula cu succes Java este necesară o configuraţie minimală constând în procesor Pentium cu frecvenţa de 166MHz, 32 MB[1] RAM şi 70MB spaţiu liber pe hard-disk .

A.2.1.1 Instrucţiuni de instalare

În această secţiune, vom arăta cum se rulează programul de instalare pentru a dezarhiva şi instala pachetul Java 2 SDK. Este posibil ca după ce Java 2 SDK a fost instalat să vi se ceară repornirea sistemului. Instalarea se face în următorii paşi:

1. În cazul în care aţi descărcat kitul de instalare de pe Internet, verificaţi dacă fişierul descărcat este complet. Fişierul `j2sdk-1_4_0-win.exe` trebuie să aibe dimensiunea de 37067134 octeţi (*bytes*) pentru ca descărcarea kitului de instalare să fie considerată completă[2];

2. În cazul în care aveţi deja instalată o versiune *Beta* sau versiunea *Release Candidate*[3] de Java 2 SDK 1.4, dezinstalaţi-o. Folosiţi pentru aceasta funcţia *Add/Remove Programs* din Windows, accesibilă din *Control Panel* (*Start -> Settings -> Control Panel*);

3. Fişierul `j2sdk-1_4_0-win.exe` reprezintă programul de instalare al Java 2 SDK. Rulaţi fişierul şi urmăriţi instrucţiunile simple care vă sunt furnizate de program;

4. Setaţi variabila sistem `PATH`. Setarea variabilei sistem `PATH` vă permite să puteţi rula convenabil executabilele Java 2 SDK (`javac.exe`, `java.exe`, `javadoc.exe` etc.) din orice director fără să fie nevoie să scrieţi întreaga cale în linia de comandă. Dacă nu setaţi variabila `PATH`, va trebui să specificaţi întreaga cale către executabil de fiecare dată când doriţi să-l rulaţi, ca în exemplul de mai jos:

[1] megabytes

[2] Atenţie totuşi la versiunea kitului. Dacă aţi ales o altă versiune a kitului Java 2 SDK (de exemplu, 1.3.0 etc.), mai mult ca sigur că dimensiunea fişierului va fi alta.

[3] versiune anterioară versiunii finale

Digitized by Google

```
c:>\jdk1.4\bin\javac MyClass.java
```

Pentru a evita o asemenea sintaxă incomodă, adăugaţi întreaga cale a directorului `jdk1.4\bin` la variabila sistem PATH. Dacă la instalare aţi ales opţiunile implicite, calea absolută în care se află executabilele Java este `c:\jdk1.4\bin`.
Setarea căii se realizează diferit, în funcţie de versiunea Windows pe care o folosiţi:
- pentru Windows NT, Windows 2000 şi Windows XP: apăsaţi *Start*, *Settings*, *Control Panel*, dublu click pe *System*. În cazul Windows NT, selectaţi *tab*-ul *Environment*; la Windows 2000 selectati *tab*-ul *Advanced* şi *Environment Variables*. Apoi, căutaţi variabila sistem `PATH` în *User Variables* sau *System Variables*. Dacă nu sunteţi sigur unde anume doriţi să adăugaţi calea către executabilele Java, adăugaţi-o la finalul variabilei `PATH` din *User Variables*. Configurarea căii conţine calea absolută a directorului în care este instalat Java (de exemplu, `c:\jdk1.4\bin`). Scrierea cu litere mari sau mici nu are vreo influenţă. În final, apăsaţi *Set*, *OK* sau *Apply*. Variabila sistem `PATH` este o succesiune de directoare separate de ";". Sistemul de operare Windows caută executabilele Java (`javac`, `java`, `javadoc`, etc.) în directoarele menţionate în cale, parcurgând variabila `PATH` de la stânga la dreapta. Este important de reţinut că în `PATH` trebuie să aveţi un singur director `bin` pentru Java SDK la un moment dat (celelalte directoare de acest fel care urmează după primul, sunt ignorate). Noua cale are efect în fiecare nouă fereastră *Command Prompt* pe care o deschideţi sau în fiecare program pe care îl executaţi de acum înainte (cum ar fi JEdit, JCreator etc.).
- pentru Windows 98, Windows 95: editaţi fişierul `autoexec.bat` (folosind de exemplu *Start* -> *Run* `sysedit`). Căutaţi secvenţa în care este definită variabila `PATH` şi adăugaţi în final calea absolută către executabilele Java. Dacă nu există nici o secvenţă de definire a variabilei `PATH`, adăugaţi-o pe o poziţie oarecare în fişier. Iată un exemplu tipic pentru setarea variabilei `PATH`:

```
PATH C:\WINDOWS;C:\WINDOWS\COMMAND;c:\jdk1.4\bin
```

Scrierea cu literele mari sau mici nu are nici o influenţă. Pentru ca setarea căii să devină funcţională, executaţi următoarea instrucţiune într-o fereastră *Command Prompt*

```
c:\> c:\autoexec.bat
```


Digitized by Google

Pentru a afla setarea curentă a căii sau pentru a vedea dacă a avut efect schimbarea, tastaţi în linia de comandă:

```
c:\> path
```

În acest moment, sistemul dumneavoastră este pregătit pentru utilizarea Java 2 SDK. Pentru a verifica acest lucru, executaţi într-un *Command Prompt* comanda:

```
c:\>java -version
```

Dacă instalarea a decurs fără probleme, va fi afişat un scurt mesaj cu date despre versiunea Java instalată. Dacă primiţi un răspuns similar cu:

```
The name specified is not recognized as an internal
or external command, operable program or batch file.
```

înseamnă că mediul Java nu a fost corect instalat sau variabila `PATH` nu are o valoare corectă.

Pentru dezinstalarea Java 2 SDK, folosiţi funcţia *Add/Remove Programs* din *Control Panel*.

A.2.2 Instalarea sub Linux

Java 2 SDK rulează de platformele Intel Pentium sau compatibile cu kernel Linux `v2.2.12` şi `glibc v2.1.2-11` sau versiuni mai recente. Aveţi nevoie de minim 32 MB RAM de memorie şi un spaţiu liber pe hard-disk de 75MB. Puteţi verifica versiunea curentă de `glibc` folosind comanda:

```
ls /lib/libc-*
```

A.2.2.1 Instrucţiuni de instalare

Pachetul Java 2 SDK, Standard Edition, v1.4.0 este disponibil în două formate de instalare:

- Un fişier binar `j2sdk-1_4_0-linux-i386.bin` care se autoextrage (autodezarhivează) şi astfel Java 2 SDK se poate instala în orice locaţie de pe disc. Pentru această metodă, consultaţi secţiunea A.2.2.2;
- Fişierul `j2sdk-1_4_0-linux-i386-rpm.bin` care conţine pachete tip RPM(Red Hat Package Manager) cu Java 2 SDK. Pentru a instala folosind această metodă, consultaţi secţiunea A.2.2.3.

Digitized by Google

Fişierele pentru ambele formate de instalare sunt cuprinse în cadrul unui fişier *shell* `script.bin` care afişează licenţa înainte de a începe instalarea.

A.2.2.2 Instalare cu autoextragere

Pentru instalare, executaţi paşii următori:

1. Verificaţi dimensiunea fişierului `j2sdk-1_4_0-linux-i386.bin`. O dimensiune de 40618207 bytes indică o descărcare corectă a kitului de instalare. Dacă mărimea fişierului nu corespunde celei de mai sus, înseamnă că fişierul a fost deteriorat în timpul descărcării. Singura alternativă este de a încerca din nou descărcarea fişierului;

2. Copiaţi `j2sdk-1_4_0-linux-i386.bin` în directorul în care doriţi să instalaţi Java 2 SDK;

3. Rulaţi `j2sdk-1_4_0-linux-i386.bin` folosind comenzile:

   ```
   chmod a+x j2sdk-1_4_0-linux-i386.bin
   ./j2sdk-1_4_0-linux-i386.bin
   ```

   Scriptul de instalare va afişa o licenţă de utilizare, după care, dacă acceptaţi condiţiile expuse în licenţă, Java 2 SDK se instalează în directorul curent;

4. Adăugaţi calea absolută către directorul `bin` rezultat în urma instalării (de exemplu, `/usr/local/jdk1.4/bin`) variabilei sistem `PATH`. Acest lucru se poate face astfel, funcţie de *shell*-ul pe care îl utilizaţi:
   - pentru *csh* sau *tcsh*:

   ```
   % set PATH=($PATH/usr/local/jdk1.4/bin)
   ```

   - pentru *sh*:

   ```
   % PATH=($PATH/usr/local/jdk1.4/bin); export $PATH
   ```

   Puteţi să adăugaţi liniile anterioare la sfârşitul fişierelor `.profile` sau `.cshrc` pentru a nu le mai scrie la fiecare *login*-are.

Pentru a verifica dacă instalarea a decurs fără probleme, deschideţi o consolă şi tastaţi comanda:

```
java -version
```


Digitized by Google

Dacă nu au fost semnalate erori pe parcursul procesului, atunci veţi vedea un mesaj cu detalii despre versiunea Java instalată. Dacă primiţi răspunsul:

```
java: Command not found
```

sau ceva similar, atunci mediul Java nu a fost bine instalat sau variabila `PATH` nu are o valoare corectă.

A.2.2.3 Instalarea folosind RPM

1. Verificaţi fişierul `j2sdk-1_4_0-linux-i386-rpm.bin`. O dimensiune de 39482030 bytes indică o descărcare corectă a kitului de instalare. Dacă mărimea fişierului nu corespunde celei de mai sus, înseamnă că fişierul a fost deteriorat în timpul descărcării. Încercaţi din nou să descărcaţi fişierul respectiv;

2. Rulaţi `j2sdk-1_4_0-linux-i386-rpm.bin` folosind comenzile:

   ```
   chmod a+x j2sdk-1_4_0-linux-i386-rpm.bin
   ./j2sdk-1_4_0-linux-i386-rpm.bin
   ```

   Scriptul va afişa o licenţă de utilizare, după care, dacă acceptaţi cerinţele licenţei, veţi obţine ca rezultat `j2sdk-1_4_0-linux-i386.rpm`, fişier ce va fi creat în directorul curent;

3. Accesaţi sistemul ca *root* folosind comanda `su` şi parola acestui cont:

   ```
   su - root
   ```

4. Dezinstalaţi J2SDK 1.4.0 Beta (în cazul în care există această versiune anterioară deja instalată pe calculatorul dumneavoastră)
   Notă: Calea implicită pentru instalarea RPM a lui J2SDK este următoarea: `/usr/java/j2sdk1.4.0`. Acesta este directorul în care au fost instalate şi *pre-release*[4]-urile versiunii 1.4 (în cazul în care acestea au fost instalate). Pentru a pregăti instalarea versiunii finale, în acest director dezinstalaţi orice versiune *pre-release* 1.4 care ar putea exista pe sistem. Dacă nu sunteţi sigur că aveţi o versiune *pre-release* 1.4, verificaţi acest lucru folosind comanda:

   ```
   rpm -query -a | grep j2sdk-1.4.0
   ```

[4] versiune anterioară versiunii finale

Digitized by Google

Comanda va afişa versiunea instalată, cu numele pachetului RPM al acesteia. După ce aţi aflat numele pachetului îl puteţi dezinstala cu una din următoarele comenzi (în funcţie de versiunea *beta* instalată în momentul curent):
- Pentru a dezinstala versiunea *Beta*:

```
rpm -e j2sdk-1.4.0-beta
```

- Pentru a dezinstala pachetul *Beta 2*:

```
rpm -e j2sdk-1.4.0-beta2
```

- Pentru a dezinstala pachetul *Beta 3*:

```
rpm -e j2sdk-1.4.0-beta3
```

5. Rulaţi comanda `rpm` pentru a instala pachetul Java 2 SDK v1.4.0:

   ```
   rpm -iv j2sdk-1_4_0-linux-i386.rpm
   ```

   Această comandă va instala Java 2 SDK pe calculatorul dumneavoastră;

6. Adăugaţi directorul `bin` rezultat în urma instalării la variabila sistem `PATH` (vezi pasul 4 de la instalarea cu autoextragere);

7. Iesiţi din contul *root*.

Puteţi verifica dacă instalarea a fost încununată de succes în acelaşi mod ca şi la instalarea cu autoextragere.

A.3 Compilarea codului sursă

Odată ce mediul de programare Java a fost instalat şi prima aplicaţie Java a fost creată cu ajutorul unui editor sau mediu IDE, ne punem problema compilării acestui prim program.

Pentru a compila o aplicaţie Java nu este suficient un simplu compilator. Pentru ca o aplicaţie să poată fi compilată, compilatorul are nevoie de bibliotecile Java standard şi eventualele biblioteci folosite de aplicaţie.

Compilatoarele sunt în general disponibile împreună cu o maşină virtuală Java, care dispune de bibliotecile Java standard şi alte utilitare. Toate aceastea

Digitized by Google

împreună formează un SDK (Standard Development Kit). Tabelul de mai jos prezintă câteva SDK-uri mai uzuale, cel mai des folosite fiind cele oferite de Sun şi IBM.

JDK	URL	OS
Blackdown v1.3	blackdown.org	Linux
IBM Developer Kit for Windows, v1.3.0	ibm.com/developerworks/ java/jdk	Windows
IBM Develorper Kit for Linux, Java 2 Technology Edition, v1.3	ibm.com/developerworks/ java/jdk	Linux
Sun Java 2 Platform, Standard Edition, v1.4	java.sun.com/j2se/1.4	Windows, Linux, Solaris

Compilatoarele incluse într-un SDK se apelează cu comanda `javac` (sau comanda `jikes` în cazul compilatorului realizat de IBM).

Pentru a compila un program Java (să presupunem că am creat un program numit `PrimulProgram.java`), trebuie să executăm într-un *Command Prompt* (consolă), în directorul în care este salvat fişierul `.java`, comanda următoare:

```
javac PrimulProgram.java
```

Dacă în urma executării acestei comenzi nu s-a afişat nici o eroare şi a fost creat un fişier `PrimulProgram.class`, atunci programul nu are greşeli de sintaxă şi putem să îl executăm.

Pentru ca bibliotecile necesare compilării unei aplicaţii să fie găsite de către compilator, acestea trebuie specificate în variabila sistem `CLASSPATH`. `CLASSPATH` (asemănător cu variabila sistem `PATH`) reprezintă o listă de directoare, arhive ZIP şi fişiere in format JAR ("jar"-urile sunt de fapt arhive standard ZIP). Obiectele listei sunt despărţite prin caracterul ';' pe Windows şi prin caracterul ':' pe Linux. De exemplu, pe Windows 98 variabila `CLASSPATH` se poate seta din `autoexec.bat` introducând comanda (evident, numele de directoare sunt prezentate orientativ):

```
set CLASSPATH=d:\api\interfata.jar;d:\api\upload\;.
```

iar pe Linux aceasta se poate seta din `/etc/profile` cu comanda:

```
export CLASSPATH=/usr/local/jars/xerces.jar:
                 /root/work/server_upload/:.
```


Digitized by Google

Setarea `CLASSPATH` se realizează într-un mod analog cu cel al variabilei `PATH` (vezi secţiunea A.2.1.1).

Compilatorul Jikes, realizat de IBM, este disponibil gratuit la următoarea adresă: `www.alphaworks.ibm.com/formula/Jikes/`. Acesta nu include bibliotecile standard şi se foloseşte de obicei împreună cu SDK-ul de la Sun, înlocuind compilatorul standard `javac`, căruia îi este superior ca viteză de compilare şi descriere a erorilor. Pentru a putea rula cu succes `jikes` trebuie ca bibliotecile standard oferite de mediul Java (aflate în `jre/lib/rt.jar`) să fie incluse în variabila sistem `CLASSPATH`.

A.3.1 Ant

Ant este o aplicaţie gen "*make*" (familiară utilizatorilor de Linux) care permite automatizarea diferitelor operaţii, cum ar fi compilarea unei aplicaţii şi pregătirea ei pentru a fi distribuită sau livrată. Ant este foarte util în cazul aplicaţiilor Java complexe, care necesită multe operaţii pentru a fi transformate din cod sursă în cod binar gata de a fi rulat. De exemplu, pentru a distribui o aplicaţie Java pentru telefoane mobile sau PDA-uri sunt necesare următoarele operaţii:

- compilarea fişierelor sursă;
- preverificarea fişierelor compilate;
- crearea unei arhive jar care să conţină fişierele sursă compilate, eventuale resurse (imagini, sunete etc.) şi alte biblioteci;
- obfuscarea[5] arhivei rezultate;
- crearea unui descriptor al aplicaţiei (fişier JAD);
- conversia arhivei într-un format acceptat de telefonul mobil sau PDA-ul pe care se va rula.

Cititorii care nu sunt familiarizaţi cu dezvoltarea de aplicaţii pe unităţi mobile nu au înţeles probabil mare lucru din paşii de mai sus. Am vrut doar să punem în evidenţă faptul că de la scrierea codului pentru o aplicaţie până la execuţia codului pe platforma destinaţie poate fi cale lungă, iar realizarea manuală a operaţiilor necesare poate fi extrem de anevoioasă. Există multe soluţii care

[5] Obfuscarea este un proces prin care se reduce dimensiunea byte-codului, folosind diverse optimizări (de exemplu, eliminarea metodelor şi atributelor care nu sunt folosite, înlocuirea getter-ilor şi setter-ilor cu atributele asupra cărora acţionează etc.).

Digitized by Google

automatizează acest proces, iar unul care se distinge prin eleganţă şi aria extrem de largă de aplicabilitate este Ant. Practic, pentru exemplul anterior, compilarea şi pregătirea aplicaţiei spre distribuţie devine o simplă rulare a comenzii `ant`.

La execuţia comenzii `ant`, se caută mai întâi în directorul curent fişierul cu numele `build.xml`. Acest fişier trebuie creat manual; el conţine mai multe *scopuri* (engl. targets) care asigură îndeplinirea anumitor sarcini. Lansând `ant` fără nici un parametru se execută scopul implicit, care de obicei este cel care asigură compilarea fişierelor sursă ale aplicaţiei. Pentru a executa alt scop, de exemplu scopul cu numele "build", se execută comanda

```
ant build
```

Iată un exemplu de fişier `build.xml`:

```
<project name="MyComplexApp" default="compile"
    basedir=".">

<target name="compile">
    <copy todir="bin">
        <fileset dir="." includes="*.properties"/>
    </copy>
    <javac srcdir="src" destdir="bin" debug="on"
        optimize="on"/>
</target>

<target name="build" depends="compile">
    <jar jarfile="MyComplexApp.jar" basedir="bin">
</target>

</project>
```

În acest exemplu, scopul `compile` copiază fişierele `.properties` (de exemplu `jndi.properties`) în directorul `bin` şi apoi compilează toate clasele din directorul `src` în directorul `bin`, păstrând structura de directoare. Scopul `build` depinde de scopul `compile`, deci când se lansează `ant build`, mai întâi se execută scopul `compile` şi dacă nu apare nici o eroare se execută scopul `build`. Scopul `build` arhivează fişierele din directorul `bin` într-o arhivă de tip jar numită `MyComplexApp.jar`.

Prezentarea Ant din cadrul acestui paragraf este mai mult decât sumară, rolul ei fiind doar de a vă deschide apetitul pentru procesarea automată. Mai multe detalii puteţi afla la adresa `http://jakarta.apache.org/ant`.

Digitized by Google

A.4 Rularea unei aplicaţii java

Execuţia unei aplicaţii Java implică existenţa (pe lângă bibliotecile folosite de aplicaţie) unei maşini virutale Java şi a bibliotecilor standard (care conţin clasele `java.lang.*`, `java.util.*`, `java.io.*` etc.). Acestea sunt incluse în SDK-uri şi JRE-uri (Java Runtime Environment). Pentru a rula o aplicaţie Java este suficient un JRE (mediu de execuţie Java) care, spre deosebire de SDK, nu conţine utilitarele pentru dezvoltarea programelor (de exemplu, `javac`), ci doar pe cele necesare execuţiei lor.

Este necesar să clarificăm nişte denumiri. Până la apariţia versiunii 1.2 a JDK-ului (Java Development Kit), denumirea Java 2 Platform Standard Edition (J2SE) nu a existat. Începând cu acea versiune, a apărut Java 2 Platform Standard Edition cu SDK 1.2 (Standard Development Kit). În continuare au apărut SDK 1.3 şi SDK 1.4 care ţin tot de Java 2 Platform Standard Edition.

Revenind, pentru a rula de exemplu `PrimulProgram.class` se foloseşte comanda următoare, executată în directorul curent:

```
java PrimulProgram
```

A.4.1 CLASSPATH

Bibliotecile necesare rulării unei aplicaţii (cu excepţia bibliotecilor standard) trebuie incluse în variabila sistem `CLASSPATH` (aşa cum am descris în paragraful A.2) sau pot fi specificate în linia de comandă, folosind optiunea `-classpath <CLASSPATH>`.

Utilitarele din SDK folosesc următoarea ordine pentru a căuta o clasă:

- clasele din `<JAVA_HOME>/jre/lib/rt.jar`, urmate apoi de clasele din arhiva `<JAVA_HOME>/jre/lib/i18n.jar;`
- clasele din `<JAVA_HOME>/jre/lib/ext/*.jar;`
- clasele din variabila `CLASSPATH.`

De exemplu dacă este căutată clasa `MyClass` din pachetul `com.mypackage` şi variabila `CLASSPATH` are valoarea
`/root/work/server_upload/:/usr/local/jars/xerces.jar`,
atunci fişierul `MyClass.class` va fi căutat astfel:

1. în arhivele din `jre/lib;`
2. în directorul `/root/work/server_upload/com/mypackage/;`

Digitized by Google

3. în arhiva /usr/local/jars/xerces.jar, subdirectorul com/my-package/.

A.5 Documentaţii java

Principala sursă de documentare pentru limbajul Java este *site*-ul oficial al companiei Sun Microsystems: java.sun.com.

Documentaţia Java 2 Platform Standard Edition se găseşte la adresa de Internet: http://java.sun.com/j2se/1.4/docs. Acolo poate fi găsită şi funcţionalitatea bibliotecilor standard (API).

Pentru a genera documentaţii în format HTML din fişiere sursă Java, se foloseşte utilitarul javadoc.

Digitized by Google

B. Convenţii de notaţie în Java. Utilitarul `javadoc`

În Mănăstire existau anumite reguli, dar Maestrul ne-a atras întotdeauna atenţia asupra tiraniei regulilor.

Anthony de Mello, O clipă de înţelepciune.

Începutul înţelepciunii constă în a desemna obiectele cu numele lor adecvate.

Proverb chinzesesc

Femeile şi pisicile vor face întotdeauna ceea ce vor, iar bărbaţii şi câinii trebuie să se relaxeze şi să se împace cu ideea.

Robert A. Heinlein

B.1 Necesitatea convenţiilor de scriere a programelor

Convenţiile de scriere sunt foarte importante pentru programatori din mai multe motive:

- Conform studiilor de specialitate, 80% din costul unei aplicaţii software este datorat întreţinerii aplicaţiei respective;

Digitized by Google

- Este foarte puţin probabil ca aplicaţia să fie întreţinută pe întreg parcursul existenţei ei doar de către cei care au dezvoltat-o;
- Convenţiile de notaţie îmbunătăţesc lizibilitatea codului, permiţând programatorilor să înţeleagă codul nou mai rapid şi mai profund;
- Dacă aplicaţia scrisă de dumneavoastră este distribuită ca produs, trebuie să vă asiguraţi că este bine realizată şi organizată.

Există deci motive foarte întemeiate pentru a utiliza convenţiile de scriere a programelor. Probabil că cititorii vor observa faptul că uneori aceste convenţii nu au fost respectate pe parcursul lucrării. Situaţia este explicabilă prin faptul că spaţiul disponibil a fost limitat, ceea ce a condus la evitarea utilizării convenţiilor mai "costisitoare" din punct de vedere al spaţiului tipografic (de exemplu, abundenţa de comentarii `javadoc`). Totuşi, sugestia noastră este să nu evitaţi folosirea acestor convenţii în programele dumneavoastră, indiferent de mărimea şi complexitatea aplicaţiilor.

B.2 Tipuri de fişiere

Java foloseşte următoarele extensii pentru fişiere:

- `.java`, pentru fişiere sursă;
- `.class`, pentru fişiere compilate (*bytecode*).

Dacă există mai multe fişiere într-un director, atunci este benefică prezenţa unui fişier `README`, care să descrie conţinutul directorului.

B.3 Organizarea fişierelor sursă `.java`

Un fişier sursă este împărţit pe secţiuni, separate prin linii goale şi, opţional, un comentariu ce identifică fiecare secţiune. Fişierele cu peste 2.000 de linii sunt incomod de parcurs şi modificat şi trebuie pe cât posibil evitate. Vom prezenta un exemplu elocvent de program Java formatat în mod corespunzător în paragraful B.10.

Fiecare fişier sursă Java conţine o singură clasă publică sau interfaţă. Când clasa publică este asociată cu alte clase sau interfeţe ne-publice acestea se pot scrie în acelaşi fişier sursă cu clasa publică, dar clasa publică trebuie să fie *prima* în cadrul fişierului.

Secţiunile unui fişier sursă Java sunt:

Digitized by Google

- comentariile de început;
- instrucţiunile `package` şi `import`;
- declaraţiile clasei şi/sau interfeţei.

B.3.1 Comentariile de început

Conţin informaţii legate de *copyright* şi o descriere a fişierului.

Informaţia *copyright* are următorul format standard:

```
/*
 * COPYRIGHT NOTICE
 * This file contains proprietary information of ...
 * Copying or reproduction without prior written
 * approval is prohibited.
 * Copyright (c) xxxxx
 */
```

Descrierea fişierului conţine observaţii legate de numele fişierului, funcţionalitatea lui, modul de utilizare a claselor sau interfeţelor care îl compun şi alte informaţii adiţionale. Comentariul trebuie să fie în stil `javadoc` (informaţii detaliate despre utilitarul `javadoc` în paragraful B.11):

```
/**
 * File name: Salut.java
 * Description : program ce afiseaza un salut de bun venit
 *               in lumea Java. Contine clasele: ...
 * Usage : standard (java Salut)
 * Version: 1.0
 * Date: 15 ianuarie 2002
 */
```

B.3.2 Instrucţiunile `package` şi `import`

Numele pachetului trebuie să apară pe prima linie necomentată din fişierul sursă şi trebuie să respecte convenţia de notare stabilită în această anexă. Imediat după aceasta, trebuie să apară declaraţiile de `import` ale claselor. De exemplu:

```
package test.capitolul4;

import java.awt.Graphics;
import java.util.*;
```


Digitized by Google

B.3.3 Declaraţiile clasei şi interfeţei

O clasă/interfaţă trebuie să fie declarată astfel (exact în această ordine):

1. comentariul de tip `javadoc` (`/** */`), în care se precizează informaţii publice despre clasă/interfaţă;

2. cuvântul cheie `class` sau `interface`, urmat de numele clasei;

3. opţional, comentariu legat de implementarea clasei/interfeţei (`/* */`). Conţine informaţii private, nenecesare pentru utilitarul `javadoc;`

4. atributele `static`. Mai întâi cele `public`, apoi cele `protected`, cele fără modificatori de acces şi, în final, cele `private`. Evident, în categoria atributelor statice intră şi constantele;

5. atributele nestatice, în aceeaşi precedenţă ca şi cele statice;

6. constructorii clasei;

7. metodele clasei. Metodele trebuie grupate din punct de vedere al funcţionalităţii, şi nu din punct de vedere al accesibilităţii. De exemplu, o metodă statică privată s-ar putea afla între două metode nestatice publice. Scopul este de a face codul cât mai uşor de citit şi înţeles. Fiecare metodă este precedată de o descriere precisă în format `javadoc`.

Documentarea este obligatorie pentru metodele publice. Metodele standard de acces la atribute (metodele de tipul `get/set`) pot fi grupate fără a avea o descriere, funcţionalitatea lor fiind binecunoscută (modifică sau returnează valoarea unui atribut al clasei). Standardul stabilit pentru descrierea metodelor este realizat în aşa fel încât descrierea să fie uşor de înţeles. Multe elemente sunt opţionale şi pot fi omise. Fiecare element de pe linie începe cu simbolul asterisk (`*`) şi se termină cu spaţiu. Dacă un element este prezentat pe mai multe linii, atunci fiecare linie începând cu a doua trebuie indentată, astfel încât fiecare linie să fie aliniată vertical cu cea precedentă.

Un exemplu arată astfel:

```
/**
 * <descriere detaliata a metodei>
 *
 * @param        <descrierea fiecarui parametru al metodei>
 * @return       <explicarea tipului de valoare returnat>
 * @exception    <explicarea fiecarei exceptii aruncate>
 * @author       <numele autorului>
 */
```


Digitized by Google

Cuvintele `@param`, `@return`, `@exception`, `@author` reprezintă etichete speciale recunoscute de `javadoc` (paragraful B.11).

Descrierea detaliată a metodei trebuie să cuprindă:

- rolul metodei;
- pre şi post-condiţii;
- efecte secundare;
- dependenţe de alte metode/clase;
- ce altă metodă ar trebui să apeleze această metodă;
- dacă metoda trebuie sau nu să fie redefinită în clasele derivate.

Secţiunea `@param` descrie tipul, clasa, constrângerile tuturor argumentelor metodei. De exemplu:

```
* @param    strSursa - stringul de intrare.
                       Nu poate fi de dimensiune 0.
```

Secţiunea `@return` descrie valoarea returnată de metodă: tipul de date returnat, domeniul de valori în care se poate afla valoarea returnată şi, dacă este posibil, informaţia de eroare returnată de metodă. De exemplu:

```
* @return   returneaza un intreg din multimea 1..n.
```

Secţiunea excepţiilor oferă o descriere a excepţiilor pe care metoda le aruncă. De exemplu,

```
* @exception ResourceNotFoundException - daca resursa
                        nu a fost gasita.
```

Deşi este evident, adăugăm un exemplu şi pentru secţiunea autor:

```
* @author   Mihai Ionescu
```

B.4 Indentarea

Codul sursă trebuie să fie indentat, folosind *tab*-uri şi nu spaţii. Indentarea cu tab-uri are avantajul că în aproape toate editoarele de texte se poate modifica nivelul de indentare selectând dimensiunea preferată pentru caracterul tab. Aceasta este în general setată la două (2) spaţii, dar se consideră acceptabil şi un *tab* de patru (4) spaţii.

Digitized by Google

B.4.1 Lungimea unei linii de cod

Trebuie evitate liniile de cod mai lungi de 80 de caractere, pentru că nu pot fi tratate corect de anumite terminale şi editoare. De asemenea, liniile care alcătuiesc comentariul `javadoc` nu trebuie să depăşească 70 de caractere.

B.4.2 "Ruperea" liniilor

În situaţia în care o anumită expresie nu este total vizibilă pe ecran (are o lungime prea mare), este indicată "ruperea" ei, pentru ca tot codul sursă să fie vizibil, fără să fie nevoie de *scroll*-area orizontală a fişierului. Regulile după care se realizează această operaţie sunt următoarele:

- "rupere" după o virgulă;
- "rupere" înaintea unui operator;
- alinierea liniei noi cu începutul expresiei la acelaşi nivel cu linia precedentă;
- dacă liniile de cod sunt confuze, indentaţi liniile noi cu 1-2 *tab*-uri.

De exemplu:

```
numeLung1 = numeLung2 * (numeLung3 + numeLung4 - numeLung5)
            + 4 * numeLung6; //ASA DA

numeLung1 = numeLung2 * (numeLung3 + numeLung4 -
                         numeLung5) + 4 * numeLung6; //ASA NU
```

B.4.3 Acoladele

Acolada de deschidere trebuie să fie pe linia următoare, aliniată vertical cu linia precendentă. Acolada de închidere trebuie să fie pe o linie separată, aliniată vertical cu acolada de deschidere.

Exemplu:

```
if (conditie)
{
    ...
}
```

Este acceptată şi deschiderea acoladei pe aceeaşi linie cu ultima instrucţiune, şi închiderea aliniată vertical cu instrucţiunea care a deschis acolada:

Digitized by Google

```
if (conditie){
    ...
}
```

Alegerea uneia dintre cele două variante este o chestiune de preferinţă personală. Prima are avantajul de a creşte lizibilitatea, dar unii se plâng că ocupă prea mult spaţiu. După cum aţi observat studiind codul din această lucrare, autorii preferă să utilizeze prima variantă.

B.4.4 Spaţierea

Liniile goale plasate adecvat pot îmbunătăţi lizibilitatea codului, împărţind codul în secţiuni logice.

Două linii vide trebuie folosite în următoarele circumstanţe:

- între secţiunile unui fişier sursă;
- între definirile claselor şi interfeţelor.

O linie vidă trebuie folosită în următoarele situaţii:

- între metode;
- între variabilele locale ale unei metode şi prima sa instrucţiune;
- înaintea unui comentariu de tip bloc sau a unui comentariu pe o singură linie (vezi secţiunea 5.1 din prezenta anexă).

Spaţiile trebuie folosite în următoarele situaţii:

- un cuvânt cheie urmat de o paranteză trebuie separat prin spaţiu de aceasta. Exemplu:

  ```
  while (true)
  {
      ...
  }
  ```

  Trebuie reţinut faptul că spaţiul nu este interpretat ca separator de către compilatorul Java atunci când se află între numele unei metode şi paranteza care deschide lista de argumente a metodei;

- un spaţiu trebuie să apară după virgulă (`,`) în lista de argumente;

Digitized by Google

- toţi operatorii binari, cu excepţia punctului (`.`) trebuie separaţi de operanzi prin spaţiu. Spaţiile nu se folosesc niciodată pentru a separa operatorii unari şi cei de incrementare/decrementare (`++`/`--`). De exemplu:

```
a = (b + c) / (d + e);
b++;
printSize(a, b, c);
```

- expresiile dintr-o instrucţiune `for` trebuie separate prin spaţiu. De exemplu:

```
for (expr1; expr2; expr3)
```

- operatorul de `cast` trebuie tot timpul urmat de spaţiu:

```
myMethod((byte) unNumar, (Object) x);
```

B.5 Comentariile

Programele Java pot avea două tipuri de comentarii: comentarii de implementare şi comentarii pentru documentaţie. Comentariile de implementare sunt de tipul celor din C++, delimitate prin `/*...*/` sau `//`. Comentariile pentru documentaţie, cunoscute sub numele de comentarii `javadoc`, sunt disponibile doar în Java şi sunt delimitate de `/**...*/`. Comentariile `javadoc` pot fi extrase şi formatate în fişiere HTML folosind aplicaţia utilitară `javadoc`.

Comentariile de implementare aduc clarificări asupra codului sursă şi sunt adresate programatorilor care lucrează sau vor lucra la scrierea codului. Comentariile de documentaţie aduc clarificări asupra rolului fiecărei clase, interfeţe, metode etc. şi sunt adresate celor care doresc să utilizeze clasele scrise de noi, fără a fi interesaţi de detaliile de implementare (codul sursă).

Comentariile conţin doar informaţii relevante pentru citirea şi înţelegerea programului. În general, este bine să se evite duplicarea informaţiei care este prezentată într-o formă clară în interiorul codului. De asemenea, este bine să se evite comentarii care ar putea deveni neactuale pe măsură ce codul sursă evoluează. Pe de altă parte, comentariile nu trebuie să fie folosite în mod abuziv. Frecvenţa mare a comentariilor reflectă uneori o calitate slabă a codului. Dacă simţiţi nevoia să comentaţi foarte des codul, este cazul să luaţi în considerare rescrierea lui pentru a-l face mai clar.

Digitized by Google

B.5.1 Comentarii de implementare

Programele pot avea patru stiluri de comentarii de implementare: în bloc, pe o singură linie, comentarii de tip *trailing* şi comentarii de tip sfârşit de linie.

- Comentarii de tipul bloc

  Comentariile de tipul bloc sunt folosite pentru a oferi descrieri ale fişierelor, metodelor, structurilor de date şi algoritmilor. Ele pot fi folosite la începutul fiecărui fişier şi înainte de fiecare metodă. Dacă sunt folosite în interiorul unei metode, comentariile de tipul bloc trebuie să fie indentate pe acelaşi nivel cu codul sursă pe care îl descriu.

  Exemplu:

```
/*
 * Acesta este un comentariu de tip bloc
 */
```

- Comentarii pe o singură linie

  Comentariile mai scurte pot să apară pe o singură linie, indentată la nivelul codului care urmează. Dacă nu poate fi scris pe o singură linie, comentariul trebuie transformat într-un comentariu de tip bloc.

  Exemplu:

```
if (conditie)
{
    /* Trateaza cazul cand conditia este adevarata. */
    ...
}
```

- Comentarii de tipul *trailing*

  Comentariile foarte scurte pot să apară pe aceeaşi linie cu codul pe care îl descriu, dar trebuie să fie indentate mai mult pentru a le separa de cod. Toate comentariile de acest tip care apar într-o porţiune de cod trebuie să aibe aceeaşi indentare.

  Exemplu:

```
if (a == 2)
{
    return true;          /* caz special, 2 este prim */
}
else
{
    return isPrime(a);    /* verifica daca a este prim*/
}
```


Digitized by Google

- Comentarii de tipul sfârşit de linie

  Delimitatorul // comentează o linie complet sau parţial.

  Exemplu:

```
if (myAge > 1)
{
    // executa o operatie
    ...
}
```

B.5.2 Comentariile de documentaţie

Platforma Java oferă o aplicaţie utilitară, denumită `javadoc`, foarte utilizată în crearea de documentaţii pentru pachetele şi clasele Java. Practic, această aplicaţie a impus un standard în materie de documentare a programelor. Comentariile de documentaţie descriu clasele, interfeţele, constructorii, metodele şi atributele. Fiecare astfel de comentariu se găseşte între delimitatorii `/** ... */`, cu specificaţia că există câte un singur comentariu pentru fiecare clasă, interfaţă sau membru al clasei.

Exemplu de comentariu de documentaţie:

```
/**
 * Clasa Vehicle ofera ...
 */
public class Vehicle
{
    ...
}
```

Comentariile de documentaţie nu trebuie poziţionate în cadrul vreunei metode sau constructor, pentru că Java asociază aceste comentarii cu prima declarare de metodă pe care o întâlneşte *după* comentariu. Amănunte despre modul de utilizare `javadoc` sunt prezentate în paragraful B.11.

B.6 Declarări

B.6.1 Numărul de declarări pe linie

Se recomandă folosirea unei singure declarări de variabilă pe o line, deoarece astfel se încurajează comentarea liniilor respective. De exemplu, varianta

```
int i; // indicele sirului
int dim; // dimensiunea sirului
```


Digitized by Google

este preferată variantei

```
int i, dim;
```

B.6.2 Iniţializarea variabilelor

Este indicat ca iniţializarea variabilelor să se facă în momentul declarării. Există situaţii în care acest lucru nu este posibil, de exemplu când valoarea iniţială a variabilei depinde de anumite calcule care trebuie mai întâi efectuate.

B.6.3 Locul de declarare a variabilelor

Cel mai bine este ca declaraţiile variabilelor să fie realizate la începutul blocului în care sunt folosite (un bloc este delimitat de acolade { }), pentru a evita posibilele confuzii de notaţie.

B.6.4 Declararea claselor şi a interfeţelor

În cazul declarării de clase sau interfeţe trebuie respectate următoarele reguli de formatare:

- nu există spaţiu între numele metodei şi paranteza "(" cu care începe lista de parametri;

- metodele sunt despărţite printr-o linie vidă

  Exemplu:

```
class Test
{
    void firstMethod()
    {
        ...
    }

    void secondMethod()
    {
        ...
    }
}
```


Digitized by Google

B.7 Instrucţiuni

B.7.1 Instrucţiuni simple

Pentru claritate, fiecare linie trebuie să conţină cel mult o instrucţiune. Tendinţa generală a programatorilor începători de a scrie mai multe instrucţiuni pe o singură linie, pentu ca astfel programele să pară mai "mici" este puerilă şi nejustificată. Un program aerisit, în care instrucţiunile se succed una sub cealaltă este mult mai uşor de parcurs şi de înţeles decât un program îmbâcsit şi înghesuit, chiar dacă acesta este de trei ori mai scurt:

```
argv++; //ASA DA
argc--; //ASA DA

argv++; argc--; //ASA NU
```

B.7.2 Instrucţiuni compuse

Instrucţiunile compuse reprezintă o listă de instrucţiuni cuprinse între acolade { }.

- instrucţiunile compuse se indentează cu un nivel faţă de instrucţiunea care le compune:

```
if (conditie)
{
    instructiune_indentata_cu_un_nivel;
}
```

- acolada de deschidere trebuie să fie pe o linie nouă aliniată vertical cu linia precedentă, iar acolada de închidere trebuie să fie pe o linie nouă, aliniată vertical cu acolada de deschidere:

```
if (conditie)
{ //aliniere verticala cu instructiunea

} //aliniere verticala cu acolada de deschidere
```

- acoladele trebuie folosite pentru toate instrucţiunile, chiar şi pentru o singură instrucţiune, atunci când fac parte dintr–o structură de tipul `if-else` sau `for`. Astfel este mai simplu de adăugat instrucţiuni, fără a genera accidental erori datorate uitării introducerii de acolade, iar codul devine mai clar.

Digitized by Google

B.7.3 Instrucţiunea `return`

Instrucţiunea `return` utilizată împreună cu o valoare nu trebuie să folosească parantezele decât dacă acestea fac valoarea returnată mai evidentă într–un anumit fel. De exemplu:

```
return;
return age;
return (dim ? dim : defaultDim);
```

B.7.4 Instrucţiunea `if-else`

Instrucţiunea `if-else` trebuie să aibă următoarea formă:

```
if (conditie)
{
    instructiuni;
}
```

sau:

```
if (conditie)
{
    instructiuni;
}
else
{
    instructiuni;
}
```

sau:

```
if (conditie)
{
    instructiuni;
}
else if (conditie)
{
    instructiuni;
}
else
{
    instructiuni;
}
```

Instrucţiunea `if` foloseşte întotdeauna acoladele. Evitaţi următoarea formă generatoare de erori logice:

```
if (conditie) //EVITATI! SE OMIT ACOLADELE.

    instructiune;
```


Digitized by Google

B.7.5 Instrucţiunea `for`

O instrucţiune `for` are următorul format:

```
for (initializare; conditie; actualizare)
{
    instructiuni;
}
```

O instrucţiune `for` vidă trebuie să fie de următoarea formă:

```
for (initializare; conditie; actualizare)
    ;
```

B.7.6 Instrucţiunea `while`

O instrucţiune `while` trebuie să fie de următoarea formă:

```
while (conditie)
{
    instructiuni;
}
```

O instrucţiune `while` vidă trebuie să fie de următoarea formă:

```
while (conditie)
  ;
```

B.7.7 Instrucţiunea `do-while`

O instrucţiune `do-while` are următoarea formă:

```
do
{
    instructiuni;
}
while (conditie);
```

B.7.8 Instrucţiunea `switch`

Instrucţiunea `switch` trebuie să arate astfel:

```
switch (conditie)
{
    case ABC:
        instructiuni;
        /* NU are break */

```


Digitized by Google

```
    case DEF:
        instructiuni;
        break;

    case XYZ:
        instructiuni;
        break;

    default:
        instructiuni;
        break;
}
```

De fiecare dată când un `case` nu are instrucţiune `break`, precizaţi acest lucru printr–un scurt comentariu. Orice instrucţiune `switch` trebuie să includă un caz `default`.

B.7.9 Instrucţiunea `try-catch`

Formatul unei instrucţiuni `try-catch` este următorul:

```
try
{
    instructiuni;
}
catch (ClasaDeTipExceptie e)
{
    instructiuni;
}
```

Numărul de instrucţiuni `catch` poate fi mai mare de 1. O instrucţiune de tipul `try-catch` poate să includă şi `finally`, care se execută indiferent dacă blocul `try` s–a executat cu succes sau nu:

```
try
{
    instructiuni;
}
catch (ClasaDeTipExceptie e)
{
    instructiuni;
}
finally
{
    instructiuni;
}
```


Digitized by Google

B.8 Convenţii de notaţie

Convenţiile de notaţie fac programele mai uşor de înţeles datorită faptului că sunt mai uşor de citit. De asemenea, ele dau informaţii despre metode, despre identificatori – de exemplu dacă este o constantă, un pachet sau o clasă – informaţii care pot fi folositoare în înţelegerea codului.

- Pachete

  Pachetele trebuie să aibă nume unice, pentru a evita posibile erori de identificare. Numele pachetului este format numai din litere mici ale alfabetului. Evitaţi folosirea majusculelor. Trebuie reţinut faptul că aplicaţiile utilitare Java fac diferenţa între literele mici şi cele mari ale alfabetului (sunt *case-sensitive*). Prefixul unui nume de pachet trebuie să fie unul dintre numele de domeniu `com`, `edu`, `gov`, `mil`, `net`, `org`, sau unul dintre codurile formate din două litere care identifică ţara, în conformitate cu standardul ISO 3166. De exemplu: `ro` pentru România, `de` pentru Germania, `fr` pentru Franţa etc. Următoarele componente ale numelui unui pachet diferă în funcţie de convenţiile fiecărei companii producătoare de software. Aceste convenţii pot specifica departamentul, proiectul etc. Numărul lor poate fi oricât de mare, dar este de preferat ca structura să nu fie prea stufoasă.

  Exemple de nume de pachete:

  ```
  com.sun.eng
  com.ibm.jdk
  ro.firmaMea.proiectul2.util
  ro.firmaMea.proiectul2.login
  ```

- Clase

  Numele claselor începe întotdeauna cu majusculă şi este format din unul sau mai multe substantive. Dacă sunt mai multe substantive, atunci prima literă a fiecărui substantiv este şi ea majusculă. Încercaţi să păstraţi aceste denumiri simple. Utilizaţi cuvintele întregi în defavoarea abrevierilor (cu excepţia cazului în care abrevierea este mai des utilizată decât numele întreg, cum este cazul `HTML` sau `URL`)

  Exemplu de nume de clase:

  ```
  class Vehicle;
  class SortedVector;
  ```

Numele claselor derivate poate fi construit, dacă este posibil, prin inserarea unui cuvânt înaintea numelui clasei de bază: din clasa `Chart` se pot deriva clasele `AreaChart`, `LineChart`;

- Interfeţe

  Numele interfeţelor respectă condiţiile impuse numelor de clase. Programatorii sunt încurajaţi să folosească prefixul `I` pentru a denumi o interfaţă, dar acest lucru nu este obligatoriu.

  Exemplu de nume de interfeţe:

```
interface IBook;
interface Disk;
```

- Metode

  Numele metodelor reprezintă acţiuni, deci vor conţine verbe. Numele metodelor începe cu literă mică, dar fiecare cuvânt, începând cu al doilea în cazul în care există, va începe cu majusculă.

  Exemplu de nume de metode:

```
void cancelOrder();
void run();
void mergeSort();
```

  Numele metodelor standard de acces la atributele unei clase (metodele de tipul `get/set`) trebuie să conţină prefixul `get` sau `set`, după cum este cazul.

  Exemplu:

```
/**
 * Returneaza valoarea atributului name din clasa.
 */
String getName();

/**
 * Modifica valoarea atributului name, atribuindu-i
 * noua valoare newName.
 */
void setName(String newName);
```

  În cazul atributelor de tip `boolean`, metodele de tip `get` trebuie să folosească prefixele `is`, `can` sau `has`.

  De exemplu, varianta:

```
//returneaza valoarea booleana student
boolean isStudent(); //ASA DA
```


Digitized by Google

este de preferat variantei:

```
boolean getStudent(); //ASA NU
```

- Variabile

  Numele variabilelor începe cu literă mică, dar fiecare cuvânt, începând cu al doilea în cazul în care există, va începe cu majusculă. Numele de variabile nu trebuie să înceapă cu linie de subliniere (_), sau simbolul dolar ($), deşi acest lucru este permis de limbajul Java.

  Numele variabilelor trebuie să fie scurte, dar cu înţeles. Numele de variabile formate dintr-o singură literă trebuie folosite doar pentru variabile temporare: `i`, `j`, `k`, `m`, `n` sunt folosite pentru variabile de tip întreg, în timp ce `c`, `d`, `e` sunt folosite pentru variabile de tip caracter.

  Exemplu de nume de variabile:

  ```
  int i;
  char c;
  float carWidth;
  ```

- Constantele

  Numele constantelor este scris în totalitate cu majuscule. Dacă sunt mai multe cuvinte în componenţa unui nume de constantă, atunci acestea sunt separate prin linie de subliniere (_).

  Exemplu de nume de constante:

  ```
  static final int MIN_WIDTH = 1;
  static final int MAX_WIDTH = 4;
  static final int GET_THE_CPU = 1;
  ```

B.9 Practici utile în programare

B.9.1 Referirea atributelor şi a metodelor statice

Evitaţi folosirea unui obiect pentru a accesa metodele şi atributele statice ale unei clase. Este de preferat să folosiţi numele clasei, punând astfel clar în evidenţă faptul că este vorba despre un membru static:

```
staticMethod(); //OK
MyClass.staticMethod(); //OK

MyClass cl = new MyClass();
cl.staticMethod(); //EVITATI
```


Digitized by Google

B.9.2 Atribuirea de valori variabilelor

Evitaţi atribuiri multiple pe aceeaşi linie de cod, ca în exemplul:

```
Clasa1.car1 = Clasa2.car2 = 'c'; //ASA NU
```

Nu folosiţi atribuiri imbricate pentru a îmbunătăţi performanţa execuţiei. Acest lucru va fi realizat de compilator:

```
d = (a = b + c) + r; //ASA NU

a = b + c; //ASA DA
d = a + r;
```

Folosiţi parantezele în cadrul expresiilor cu mulţi operatori, pentru a evita probleme de precedenţă a operatorilor:

```
if (a == b && c == d) //ASA NU

if ((a == b) && (c == d)) //ASA DA
```

Adăugaţi comentarii speciale anumitor părţi de cod despre care ştiţi că sunt susceptibile la erori sau nu funcţionează deloc.

B.10 Exemplu de fişier sursă Java

Fişierul Java listat în continuare este un exemplu de program Java care respectă convenţiile de programare prezentate în această anexă:

```
/*
 * COPYRIGHT NOTICE
 * This software is the confidential and proprietary
 * information of MySoftware Ltd.
 *
 * Copyright (c) 2002 MySoftware Ltd.
 * Silicon Alley 4B, Brasov, Romania
 * All rights reserved.
 */

package com.myPackage.vehicles;

import java.util.Vector;

/**
 * FileName: Car.java
 * Description: defineste o masina
 *
 * @version 1.00    15 Ianuarie 2002
```


Digitized by Google

```java
 * @author  Ionescu Mihai
 */
public class Car extends Vehicle
{
    /* Comentariu de implementare a clasei. */

    /** Comentariu de documentatie pentru classVar1. */
    public static int classVar1;

    /**
     * Comentariul de documentatie pentru variabila classVar2
     * este mai lung de o linie.
     */
    private static Object classVar2;

    /** Comentariu de documentatie pentru instanceVar1. */
    public Object instanceVar1;

    /** Comentariu de documetatie pentru instanceVar2. */
    protected int instanceVar2;

    /** Comentariu de documentatie pentru instanceVar3. */
    private Object[] instanceVar3;


    /**
     * Comentariu de documentatie pentru constructorul clasei
     */
    public Car()
    {
        // ... implementarea constructorului...
    }

    /**
     * Comentariu de documentatie pentru metoda doSomething.
     */
    public void doSomething()
    {
        // ...implementarea metodei...
    }

    /**
     * Comentariu de documentatie pentru metoda doSomethingElse.
     * @param someParam descrierea parametrului
     */
    public void doSomethingElse(Object someParam)
    {
        // ...implementarea metodei...
    }
}
```


Digitized by Google

B.11 Utilitarul `javadoc`

`Javadoc` este o aplicaţie utilitară oferită de Sun Microsystems împreună cu JDK pentru a genera documentaţie API[1] în format HTML pornind de la comentariile de documentaţie din cadrul fişierelor sursă. `Javadoc` este inclus în distribuţia J2SDK, sub forma unei aplicaţii cu acelaşi nume şi se află în directorul `bin` al distribuţiei. Utilitarul nu poate fi obţinut separat, ci doar ca parte integrantă a J2SDK.

`javadoc` este foarte bine documentat de către creatorii săi. Lucrarea de faţă va prezenta în cele ce urmează, doar informaţii absolut necesare pentru a putea folosi acest utilitar. Mult mai multe informaţii despre `javadoc` se pot găsi fie în cadrul documentaţiei J2SDK, fie la adresa următoare:
`http://java.sun.com/j2se/javadoc/index.html`.

B.11.1 Sintaxa `javadoc`

Sintaxa generală `javadoc` arată astfel:

```
javadoc [optiuni] [nume-pachete] [fisiere-sursa]
           [@nume-fisiere]
```

Iată descrierea celor patru parametri opţionali ai comenzii:

- `optiuni` reprezintă opţiunile utilitarului şi vor fi detaliate pe parcursul acestei anexe;
- `nume-pachete` reprezintă numele pachetelor pentru care se crează documentaţia. Numele sunt separate prin spaţiu, ca în exemplul `java.io java.util java.lang`. Fiecare pachet care va fi documentat trebuie specificat separat. De reţinut faptul că `javadoc` foloseşte opţiunea numită `-sourcepath` pentru a căuta aceste pachete. Transmiterea ca argument a unui nume de pachet implică procesarea de către `javadoc` a tuturor fişierelor `.java` din directorul corespunzător pachetului respectiv, chiar şi dacă fişierele `.java` reprezintă exemple de utilizare a anumitor clase sau, pur şi simplu, sunt clase care nu fac parte din pachetul

[1] API = Application Program Interface. Acest termen desemnează metoda specifică prescrisă de sistemul de operare al unui calculator sau de o anumită aplicaţie prin care un programator care scrie o aplicaţie poate interacţiona cu sistemul de operare sau aplicaţia respectivă. În cazul nostru, documentaţia API reprezintă o descriere în format HTML a claselor, interfeţelor, metodelor, atributelor etc. din cadrul pachetelor Java, care are rolul de a descrie rolul acestora şi modul în care pot fi utilizate de un programator.

Digitized by Google

respectiv. Acest lucru are loc din cauza faptului că `javadoc` nu parsează fişierele `.java` în căutarea unei declaraţii de pachet (`package`);

- `fisiere-sursa` reprezintă numele fişierelor sursă, separate prin spaţiu, fiecare dintre fişiere putând fi precedate de locaţia lor: `Button.java`, `awt/Graphics*.java`. Calea care precedă numele fişierului sursă determină unde va fi căutat acesta de `javadoc`. `Javadoc` *nu* foloseşte `-sourcepath` pentru a căuta fişierele sursă. Exemplul precedent arată totodată că este permisă folosirea de *wildcard*-uri (`Graphics*.java`). Acestea uşurează uneori foarte mult lucrul celui care realizează documentaţia, deoarece mai multe fişiere pot fi precizate printr-o singură comandă (de exemplu, `Graphics*.java` cuprinde denumirile tuturor fişierelor a căror denumire începe cu `Graphics`, este urmată de orice succesiune de caractere şi au extensia `.java`: `Graphics.java`, `GraphicsStandard.java`, `GraphicsTemplate.java` etc.). Clasele sau interfeţele pentru care se generează documentaţia completă prin rularea `javadoc` poartă numele de clase documentate;

- `@nume-fisiere` reprezintă fişiere care conţin numele de pachete sau de fişiere care vor fi documentate. Ordinea pachetelor şi a claselor din cadrul fişierelui nu are importanţă, dar trebuie să apară câte una pe linie. Această opţiune este în special utilă pentru a evita apariţia unor liste foarte "stufoase" de parametri reprezentând numele de pachete şi fişierele sursă.

B.11.2 Descrierea utilitarului

`Javadoc` parcurge declaraţiile şi comentariile de documentaţie dintr-un set de fişiere Java şi crează un set corespondent de pagini `HTML`, descriind (implicit) clasele `public` şi `protected`, clasele interioare (*engl.* inner classes), intefeţele, constructorii, metodele şi atributele. Utilitarul poate fi folosit pe pachete întregi, pe fişiere sursă separate, dar şi în variantă combinată, pe pachete şi fişiere. În primul caz, argumentul utilitarului `javadoc` este o serie de nume de pachete, în timp ce în al doilea caz, argumentul este reprezentat de o serie de nume de fişiere sursă `.java`.

Pentru a putea fi utilizat, `javadoc` are nevoie de compilatorul Java. `Javadoc` utilizează `javac` pentru a compila declaraţiile din fişierele sursă, ignorând implementarea membrilor claselor respective. În acest fel, `javadoc` construieşte o bogată reprezentare internă a claselor, inclusiv ierarhia de clase şi relaţiile dintre clase, create prin utilizarea lor. Folosind această reprezentare, `javadoc` generează documentaţia `HTML`. Pe lângă aceste informaţii, `javadoc`

Digitized by Google

mai foloseşte şi informaţiile oferite de programator în cadrul comentariilor de documentaţie din fişierele sursă.

Când crează structura internă a claselor, `javadoc` încarcă toate clasele referite, cu alte cuvinte clasele sau interfeţele care sunt explicit referite în definirea (implementarea) claselor/interfeţelor documentate. Exemple de referire sunt tipul de valoare returnată, tipurile de parametri, tipul către care se realizează o operaţie de `cast`, clasa extinsă, interfaţa implementată, clasele importate, clasele folosite în definirea corpurilor metodelor, etc. Pentru a funcţiona corect, `javadoc` trebuie să fie în măsură să găsească toate clasele referite. Faptul că `javadoc` utilizează compilatorul Java, `javac`, ne asigură de faptul că documentaţia `HTML` obţinută corespunde exact cu implementarea propriu-zisă.

Conţinutul şi formatul documentaţiei generate de `javadoc` pot fi modificate conform preferinţelor folosind o componentă denumită *doclet*. Utilitarul `javadoc` are în mod implicit o astfel de componentă, denumită *doclet standard*, care generează documentaţia API în format `HTML`. Doclet-ul standard poate fi înlocuit cu altul care generează documentaţie în format `PDF`, `XML`, `MIF`, `RTF` sau alte formate. Există diverse doclet–uri create de Sun Microsystems, dar şi de alte companii. Sun oferă trei doclet–uri: *doclet-ul standard*, prezentat anterior, *doclet-ul MIF*, care generează documentaţia API în format `MIF` (folosit de Adobe FrameMaker), `PDF`, `PS` sau `RTF`, şi *doclet–ul DocCheck*, pentru verificarea corectitudinii comentariilor de documentaţie din fişierele sursă.

Lista doclet-urilor pe care le puteţi folosi include şi următoarele aplicaţii, realizate de diverse alte companii:

- *Dynamic Javadoc* (sau DJavadoc), ce generează `HTML` în format dinamic (DHTML);

- *RTFDoclet*, care generează documente în format `RTF`, care pot fi vizualizate cu MS Word;

- *DocLint*, care verifică corectitudinea comentariilor de documentaţie[2].

Totuşi, dacă sunteţi de părere că aceste doclet–uri nu vă sunt de folos, puteţi să vă implementaţi propriul dumneavoastră doclet.

Implicit, `javadoc` utilizează doclet-ul standard şi în majoritatea situaţiilor vă veţi mulţumi să îl utilizaţi pe acesta. Schimbarea acestuia cu unul la alegere se face prin specificarea opţiunii `-doclet` la apelul `javadoc`.

[2]Probabil că vă întrebaţi cum poate fi o documentaţie incorectă. După cum vom vedea în continuare, comentariile `javadoc` pot să conţină şi anumite cuvinte cheie (etichete) care impun o anumită sintaxă. Corectitudinea comentariilor de documentaţie se referă la utilizarea corespunzătoare a acestor cuvinte cheie, şi nicidecum la corectitudinea conţinutului.

Digitized by Google

După cum precizam anterior, `javadoc` generează documentaţia API pe baza fişierelor sursă `.java`. Opţional, acestora li se mai pot adăuga şi alte tipuri de fişiere:

- fişiere de comentare a pachetului. Fiecare pachet poate avea propriul lui comentariu de documentare, în care să fie precizate informaţii valabile pentru întreg pachetul. Pentru a realiza acest lucru, trebuie creat un fişier `HTML` cu numele `packages.html`, localizat în acelaşi director cu fişierele sursă `.java` care formează pachetul respectiv. `Javadoc` caută automat în fiecare director un fişier cu acest nume şi îl cuprinde în documentaţie;

- fişiere de comentare per ansamblu. Există posibilitatea de a crea un comentariu la nivel de aplicaţie sau pentru mai multe pachete simultan. Crearea unui astfel de comentariu se face prin intermediul unui fişier denumit `overview.html`. Fişierul poate fi plasat oriunde, pentru că este unic în cadrul unei documentaţii generate, dar este de preferat să fie localizat în directorul rădăcină al fişierelor sursă `.java`. Ca şi în cazul anterior, acest fişier trebuie scris în `HTML`;

- fişiere de alte tipuri (imagini, exemple de programe Java etc.), care sunt doar copiate de `javadoc` în directorul destinaţie. Aceste fişiere sunt necesare în cazul în care, de exemplu, comentariul de documentaţie din fişierul sursă face referire la o imagine etc.

Implicit, rezultatul generării documentaţiei este o structură de directoare, deoarece `javadoc` foloseşte doclet-ul standard, care generează `HTML`. Directoarele respective conţin fişierele `HTML` generate propriu-zis de `javadoc` şi eventualele fişiere (imagini, exemple de programe) care au fost doar copiate de `javadoc`. Pentru vizualizarea documentaţiei generate, este de ajuns să aveţi un *browser* de Internet (de exemplu, Internet Explorer sau Netscape Navigator) şi să accesaţi pagina `index.html` din rădăcina directorului în care se află documentaţia generată. Accesarea acestei pagini oferă posibilitatea de a vedea rezultatul eforturilor de documentare a fişierelor Java. Documentaţia este prezentată sub forma a două *frame*-uri HTML (dacă nu a fost generată pentru pachete) sau sub forma a trei *frame*–uri HTML (în cazul pachetelor):

```
---------------
| C | Detaliu |
|   |         |
|   |         |
---------------
```


Digitized by Google

C = lista claselor pentru care s-a realizat documentaţia
Detaliu = detaliile unei clase selectate în frame-ul din stânga

```
---------------
| P | Detaliu |
|---|         |
| C |         |
---------------
```

P = lista pachetelor pentru care s-a realizat documentaţia
C = lista claselor din pachetul selectat în frame-ul din stânga sus
Detaliu = detaliile unei clase selectate în frame-ul din stânga jos

Dacă pentru generarea documentaţiei nu a fost folosit doclet-ul standard, atunci formatul documentaţiei generate este altul, pentru vizualizarea ei fiind necesare, în mod firesc, alte aplicaţii.

Implicit, javadoc transformă comentariile în cod HTML, de aceea comentariile în sine pot conţine cod HTML, pentru a permite o formatare mai bună. Prima propoziţie a comentariului trebuie să reprezinte o descriere succintă dar completă a elementului comentat, deoarece această propoziţie este folosită de javadoc pentru a crea un rezumat al descrierii clasei, interfeţei, metodei sau pachetului respectiv. Se consideră sfârşitul propoziţiei simbolul punct (.), urmat de spaţiu, *tab* sau apariţia unui etichete speciale suportată de javadoc (detalii despre etichetele javadoc în secţiunea 11.3). Aşadar, evitaţi comentariile de genul:

```
/**
 * Modifica nota acordata de prof. unui elev.
 */
```

Dacă doriţi să folosiţi totuşi exprimarea anterioară, atunci apelaţi la următorul truc:

```
/**
 * Modifica nota acordata de prof. unui elev.
 */
```

B.11.3 Etichete javadoc

În cadrul comentariilor de documentaţie din interiorul fişierelor sursă, javadoc suportă anumite etichete, care sunt tratate special. Fiecare etichetă începe cu semnul @.

Lista completă a etichetelor javadoc este formată din următoarele elemente:

Digitized by Google

- `@author`
- `@deprecated`
- `@exception`
- `@param`
- `@return`
- `@see`
- `@serial`
- `@serialData`
- `@serialField`
- `@since`
- `@throws`
- `@version`
- `{@docRoot}`
- `{@link}`

Unele dintre acestea au fost deja utilizate în cadrul paragrafului B.3.3, la declararea claselor şi a interfeţelor, altele vor fi prezentate în continuare.

`@deprecated` adaugă în documentul generat un comentariu care arată că elementul respectiv (clasă, interfaţă, metodă etc) este învechit şi că nu ar mai trebui folosit (chiar dacă ar putea să funcţioneze). Totodată, trebuie precizat elementul care ar trebui folosit în locul celui demodat. Mod de utilizare: `@deprecated` *comentariu*.

`@see` adaugă informaţii ajutătoare, de genul "vezi metoda X". Informaţia apare la secţiunea *See also* a documentaţiei generate, putând fi un simplu text sau un *link* către alte elemente. Poate fi utilizat în următoarele forme:

- `@see "comentariu"`, afişează text simplu
- `@see <a href="URL-ul">Numele link-ului< /a>`, afişează un *link* HTML

Digitized by Google

- `@see pachet.clasa#membru numeLink`, afişează un link `HTML` către documentaţia generată pentru membrul care aparţine clasei din pachetul specificat.

`@since` afişează un text care indică versiunea aplicaţiei software care a prezentat pentru prima dată elementul respectiv. De exemplu, `@since 1.3` denotă faptul că elementul a apărut în versiunea 1.3 a aplicaţiei.

`@version` este utilizat în general cu argumentul `"%I%, %G%"`, care este convertit de `javadoc` în ceva similar cu `"1.39, 02/28/99"` (luna/zi/an), adică versiunea şi data trecerii la versiunea respectivă.

`{@link pachet.clasa#membru numeLink}` se aseamănă cu eticheta `@see`, cu deosebirea că link-ul creat nu este afişat în secţiunea *See also*, ci chiar în cadrul textului în care este folosit.

B.11.4 Opţiunile comenzii `javadoc`

Opţiunile comenzii `javadoc` sunt destul de numeroase. Pentru a vedea lista completă a tuturor acestor opţiuni, executaţi

`javadoc`

în linia de comandă. Vom prezenta în continuare cele mai uzuale opţiuni:

`-overview` *`caleCatreFisier`*

`Javadoc` foloseşte această opţiune pentru a prelua comentariul de ansamblul al aplicaţiei din fişierul specificat (`overview.html`). Calea către fişier este relativă în funcţie de opţiunea `-sourcepath`.

`-public`

Generează documentaţie doar pentru clasele şi membrii `public`.

`-protected`

Afişează doar clasele şi membrii `public` şi `protected`. Această opţiune este implicită.

`-package`

Afişează doar pachetele, clasele şi membrii `public` şi `protected`.

`-private`

Afişează toate clasele şi toţi membrii.

`-doclet` *`numeClasa`*

Este specificată clasa care execută doclet-ul folosit în generarea documentaţiei. Calea către clasa de start a doclet–ului este specificată prin intermediul opţiunii `-docletpath`, urmată de calea respectivă.

`-sourcepath` *`listaCaiFisiereSursa`*

Este specificată calea către fişierele sursă (`.java`), atunci când numele de pachete sunt folosite ca parametri. De exemplu, dacă se doreşte documenta-

Digitized by Google

rea unui pachet numit `com.mypackage`, ale cărui fişiere sursă sunt localizate la locaţia `c:\user\src\com\mypackage\*.java`, atunci comanda ce trebuie executată arată astfel:

`c:\>javadoc -sourcepath c:\user\src com.mypackage`

`-classpath` *`listaCaiClase`*

Este specificată calea unde `javadoc` va căuta clasele referite (pentru detalii despre clasele referite, vezi secţiunile anterioare din această anexă).

`-J`*`optiune`*

Lansează comanda `java` cu opţiunea specificată de argumentul *`optiune`*. De exemplu, pentru aflarea versiunii utilizate de `javadoc`, se execută comanda:

`c:\> javadoc -J-version`

Trebuie reţinut faptul că nu există nici un spaţiu între opţiunea `J` şi argumentul *`optiune`*.

Opţiunile prezentate anterior pot fi utilizate de `javadoc` indiferent de doclet-ul folosit în generarea documentaţiei. Implicit, `javadoc` utilizează doclet-ul standard, care generează documentaţie în format `HTML`. Acest doclet oferă un alt set de opţiuni, care vine în completarea celui anterior. Iată câteva dintre aceste opţiuni:

`-d`

Este specificat directorul destinaţie, unde `javadoc` va salva documentaţia `HTML` generată. Dacă această opţiune nu este prezentă, atunci documentaţia va fi creată în directorul curent.

`-version`

Include textul asociat cu eticheta `@version` în documentaţia `HTML` generată

`-author`

Include textul asociat cu eticheta `@author` în documentaţia `HTML` generată

`-splitindex`

Împarte alfabetic fişierul de index al documentaţiei în mai multe fişiere, câte unul pentru fiecare literă.

`-windowtitle` *`text`*

Este specificat titlul care va fi plasat în tag-ul `<title>` din `HTML` (denumirea ferestrei *browser*-ului). Acest titlu devine titlul ferestrei de *browser*, vizibil în colţul din stânga sus al ferestrei.

`-doctitle` *`text`*

Este specificat titlul documentului.

`-header` *`text`*

Este specificat textul care va fi plasat la începutul fiecărui fişier de documentaţie generat.

Digitized by Google

`-footer` *`text`*

Este specificat textul care va apărea în dreapta meniului de navigare din josul paginii.

`-bottom` *`text`*

Este specificat textul care va apărea la finalul fiecărui fişier generat. În general, aceste note de subsol conţin informaţii legate de copyright.

`-nodeprecated`

Elementele demodate (pachete, clase, membrii etc.) sunt omise la generarea documentaţiei.

B.11.5 Exemple simple de utilizare `javadoc`

Vom prezenta în final câteva exemple simple de folosire a utilitarului `javadoc`. Utilitarul poate fi rulat pe clase individuale sau pe pachete întregi de clase.

- Crearea documentaţiei pentru unul sau mai multe pachete

  Pentru a documenta un pachet, fişierele sursă din acel pachet trebuie să existe într-un director cu acelaşi nume ca şi pachetul. Dacă pachetul este creat din mai multe nume, separate prin simbolul punct (.), fiecare nume reprezintă un director diferit. Astfel, toate clasele din pachetul `java.awt` trebuie să existe în directorul `java\awt\`.
  Presupunem că directorul `c:\home\src\java\awt\` este directorul în care sunt create fişierele sursă iar directorul destinaţie pentru documentaţia generată este `c:\home\html`. Vom păstra această convenţie pentru toate exemplele de acest gen din anexa curentă.
  Documentaţia se poate crea în două moduri:

  – din directorul corespunzător pachetului:

  ```
  c:\> cd c:\home\src
  c:> javadoc -d c:\home\html java.awt
  ```

  – din orice director:

  ```
  c:\> javadoc -d c:\home\html -sourcepath
                      c:\home\src java.awt
  ```

- Crearea documentaţiei pentru una sau mai multe clase

  Analog cazului 1, avem două modalităţi disponibile:

Digitized by Google

– din directorul cu fişiere sursă:

```
c:\> cd c:\home\src\java\awt
c:> javadoc -d c:\home\html Canvas.java
                            Graphics*.java
```

Se generează documentaţia pentru clasa `Canvas` şi pentru toate clasele al căror nume începe cu `Graphics`. Documentul `HTML` generat nu va mai avea trei frame-uri ca în cazul pachetelor, ci două, deoarece generarea s-a realizat pe clase şi nu pe pachete, situaţie în care al treilea frame (cel cu listarea pachetelor) este inutil.

– din orice director:

```
c:\> javadoc -d c:\home\html
       c:\home\src\java\awt\Canvas.java
       c:\home\src\java\awt\Graphics*.java
```

- Crearea documentaţiei pentru pachete şi clase simultan

  Combinarea celor două cazuri descrise anterior, conduce la:

```
c:\> javadoc -d c:\home\html -sourcepath
       c:\home\src java.awt
       c:\home\src\java\applet\Applet.java
```

- Crearea documentaţiei prin specificarea în linie de comandă a unui fişier

  Pentru a scurta sau simplifica o comandă `javadoc`, se pot specifica unul sau mai multe fişiere care conţin separat nume de fişiere sursă sau pachete, câte unul pe linie. La execuţia comenzii, numele fişierelor trebuie precedat de simbolul `@`.

  De exemplu, fie fişierul `packages`, cu următorul conţinut:

```
com.package1
com.package2
com.package3
```

  Se poate rula `javadoc` astfel (`apidocs` reprezintă numele directorului de destinaţie al documentaţiei generate, iar `home\src` constituie directorul în care se află fişierul `packages`, dar şi directorul `com`, cu subdirectoarele `package1`, `package2`, `package3`):

Digitized by Google

```
c:\> cd c:\home\src
c:> javadoc -d apidocs @packages
```

- Salvarea mesajelor de eroare generate de execuţia `javadoc` într–un fişier separat (valabil doar pe Windows NT)

  Pentru a redirecta eventualele mesaje de eroare generate de execuţia comenzii `javadoc`, se poate utiliza comanda:

  ```
  c:\> javadoc -d docs java.lang >log.std 2>log.err
  ```

  sau, forma restrânsă:

  ```
  c:\> javadoc -d docs java.lang >javadoc.log 2>&1
  ```

  Prima variantă redirecţionează mesajele standard în fişierul `log.std`, iar pe cele de eroare în `log.err`, în timp ce în al doilea caz, ambele tipuri de mesaje sunt redirecţionate către fişierul `javadoc.log`.

- Utilizarea din alte programe a comenzii `javadoc`

  Se poate folosi o secvenţă de cod de genul:

  ```
  void metoda()
  {
     String[] javadocArguments = {
                                   "-sourcepath",
                                   "c:\home\src",
                                   "-d",
                                   "c:\home\html",
                                   "java.awt"
                                 };
     com.sun.tools.javadoc.Main.main(javadocArguments);

     //continuare cod...
  }
  ```

  Dezavantajul unei astfel de utilizări este că metoda `main` nu returnează nimic, de aceea nu se poate determina dacă `javadoc` s-a executat cu succes sau nu.

Digitized by Google

C. Definirea pachetelor Java. Arhive `jar`

Simplificaţi lucrurile atât cât este posibil, dar nu mai mult.

Albert Einstein

C.1 Prezentare generală a pachetelor Java predefinite

Mediul de programare Java oferă o serie de clase predefinite pentru a veni în întâmpinarea programatorilor care doresc să realizeze aplicaţii la standarde profesionale. Scopul pentru care aceste clase au fost implementate de creatorii limbajului Java, a fost acela de a uşura pe cât posibil munca programatorilor, care nu mai sunt obligaţi în acest fel "să reinventeze roata". Programatorii folosesc aceste clase, dar îşi pot defini pe baza lor şi propriile lor clase, specifice aplicaţiilor pe care le realizează.

Mulţimea claselor oferite de limbajul Java poartă numele de Java 2 API (Application Programming Interface). API-ul Java oferă facilităţi foarte puternice, des utilizate de programatori. Putem spune cu certitudine că orice program Java utilizează din abundenţă API-ul Java. Dată fiind importanţa lui, este evident că documentaţia API-ului capătă şi ea o însemnătate deosebită. Documentaţia poartă numele Java 2 API Specification şi poate fi găsită în cadrul documentaţiei J2SDK.

Pentru o mai bună organizare, clasele oferite de limbajul Java au fost împărţite în pachete, având drept criteriu de departajare funcţionalitatea oferită. Platforma Java este deosebit de solidă şi oferă diverse facilităţi în diferite di-

Digitized by Google

recţii, cum ar fi: *applet*-uri, aplicaţii de tip *desktop* folosind *AWT*, aplicaţii *Swing* bazate pe *JFC*, RMI (Remote Method Invocation), securizarea aplicaţiilor, utilizarea bazelor de date, internaţionalizarea aplicaţiilor, crearea de arhive *JAR* şi *ZIP*, suport pentru *CORBA*, etc. Multitudinea de direcţii în care s-a dezvoltat Java este observabilă şi în numărul mare de pachete pe care îl pune la dispoziţia programatorilor: peste 70 de pachete. Numărul este impresionant şi devine evident faptul că nu putem prezenta informaţii despre fiecare pachet în parte. Totuşi, în cele ce urmează vom descrie cel mai des utilizate pachete, împreună cu câteva dintre cele mai reprezentative clase conţinute.

Iată o listă a celor mai des utilizate pachete (în cadrul acestei lucrări sunt folosite doar clase din cadrul primelor cinci pachete):

- `java.lang`, care oferă clase fundamentale pentru limbajul Java;
- `java.util` conţine clase pentru utilizarea colecţiilor, pentru manevrarea datelor calendaristice şi a timpului, pentru internaţionalizarea aplicaţiilor etc.;
- `java.io`, care oferă modalităţi de citire/scriere a datelor prin intermediul fluxurilor de date, a serializării sau a fişierelor;
- `java.math`, care oferă clase specializate în calcul matematic;
- `java.text`, care oferă clase pentru manevrarea textului, a datelor calendaristice, a timpului şi a mesajelor într-o manieră independentă de limba utilizată, fie ea engleză, franceză, română etc.;
- `java.net`, care pune la dispoziţia programatorilor clase pentru implementarea aplicaţiilor de reţea;
- `java.util.jar`, care oferă clase pentru citirea şi scrierea fişierelor în format JAR (Java Archive), bazat pe formatul standard ZIP;
- `java.util.zip`, care oferă clase pentru citirea şi scrierea de arhive în format ZIP sau GZIP.

Iată şi câteva dintre cele mai reprezentative clase pentru fiecare dintre aceste pachete:

- `java.lang`
  - `Boolean`, `Byte`, `Character`, `Double`, `Float`, `Integer`, `Long`, `Short`, asociate tipurilor primitive respective: `byte`, `char`, `double` etc.;

Digitized by Google

- `Math`, asigură realizarea unor operaţii aritmetice elementare: modulul, funcţii trigonometrice, exponenţiale, maximul şi minimul a două numere, aproximări etc.;
- `String`, `StringBuffer`, pentru administrarea şirurilor de caractere;
- `System`, oferă facilităţi de acces la fluxurile standard de intrare (`in`), ieşire (`out`) şi eroare (`err`) etc.;
- `Thread`, asigură facilităţi pentru lucrul cu fire de execuţie în Java.

• `java.util`

- `Calendar`, pentru administrarea datelor calendaristice;
- `Date`, reprezintă o dată calendaristică în timp, cu precizie la nivel de milisecunde;
- `Hashtable`, pentru reprezentarea tabelelor de repartizare;
- `LinkedList`, pentru administrarea listelor înlănţuite;
- `Locale`, pentru reprezentarea unei zone politice, geografice sau culturale;
- `ListResourceBundle`, `PropertyResourceBundle` sau `ResourceBundle`, pentru reprezentarea colecţiilor de resurse;
- `Stack`, pentru administrarea unei stive (listă în format LIFO, LastIn-FirstOut);
- `Vector`, implementează un şir de obiecte de dimensiune variabilă.

• `java.io`

- `BufferedReader`, citeşte text de la un flux de intrare a datelor, depunând informaţia citită într-un *buffer* pentru a realiza o citire eficientă a caracterelor, şirurilor, stringurilor;
- `BufferedWriter`, scrie text către un flux de ieşire a datelor, depunând informaţia scrisă într-un *buffer* pentru a realiza o scriere eficientă a caracterelor, şirurilor, stringurilor;
- `File`, care asigură gestionarea fişierelor, inclusiv a locaţiilor unde acestea se găsesc;
- `FileInputStream`, asigură citirea de date sub forma unor şiruri de *bytes* dintr-un fişier specificat;

Digitized by Google

  - `FileOutputStream`, asigură scrierea de şiruri de *bytes* într-un fişier specificat;

- `java.math`

  - `BigDecimal`, `BigInteger`, care permit calculul cu numere foarte mari, care nu ar "încape" în reprezentarea obişnuită a numerelor în Java.

- `java.text`

  - `ChoiceFormat`, permite ataşarea de formate de mesaje în funcţie de diverse valori ale unor variabile;
  - `DateFormat`, asigură formatarea datelor calendaristice şi a timpului într-o manieră independentă de limba utilizată;
  - `MessageFormat`, permite formatarea mesajelor independent de limbă;
  - `NumberFormat`, permite formatarea numerelor, a procentelor şi a sumelor de bani independent de limbă;

- `java.net`

  - `HttpURLConnection`, oferă facilităţi pentru conectarea prin reţea la un server HTTP şi adresarea unei cereri serverului respectiv;
  - `InetAddress`, pentru administrarea unei adrese IP (Internet Protocol);
  - `URL`, pentru administrarea unei adrese URL (Uniform Resource Locator), care indică o resursă Internet.

- `java.util.jar`

  - `JarFile`, pentru administrarea unui fişier JAR;
  - `JarInputStream`, oferă facilităţi de citire a conţinutului unui fişier JAR;
  - `JarOutputStream`, oferă facilităţi de scriere a unui fişier JAR.

- `java.util.zip`

  - `ZipFile`, pentru administrarea unei arhive ZIP;

Digitized by Google

- `ZipInputStream`, oferă facilităţi de citire a conţinutului unei arhive ZIP;
- `ZipOutputStream`, oferă facilităţi de scriere a unei arhive ZIP.

Dintre pachetele prezentate, două se detaşează prin gradul foarte mare de utilizare, indiferent de natura aplicaţiei care se crează. Este vorba despre pachetele `java.lang` şi `java.util`. Aşa cum am precizat, pachetul `java.lang` este importat implicit în orice aplicaţie Java, drept urmare programatorul poate folosi clasele sale fără a fi nevoie să le importe explicit.

C.2 Definirea de pachete de către utilizator

Secţiunea de faţă prezintă un exemplu simplu, dar detaliat, de creare şi utilizare a propriilor pachete. Mediul de programare Java oferă spre utilizare programatorului, o mulţime de pachete împreună cu clasele aferente (de exemplu, `java.util`, `java.math` etc.). Dar pentru că această facilitate nu este întotdeauna suficientă, mediul Java permite definirea propriilor pachete. Cu alte cuvinte, programatorului Java îi este permis să îşi organizeze clasele definite de el în pachete proprii.

Exemplul ales utilizează trei pachete:

- `com.test.masini`
- `com.test.posesori`
- `com.test`

Primul pachet, `com.test.masini`, va îngloba clase cu funcţionalitate apropiată, reprezentând diverse tipuri de maşini. Aşadar, vom avea clasele `Dacia` şi `Ferrari`. Al doilea pachet, `com.test.posesori`, va cuprinde clase care descriu posesorii automobilelor din primul pachet. Printre aceste clase se numără: `Persoana`, clasă asociată unei persoane fizice şi `Firma`, clasă asociată unei persoane juridice (firme).

Programul principal, cu alte cuvinte clasa `ExempluPachete`, se va afla în pachetul `com.test` şi va utiliza clase din celelalte două pachete. Pentru că definirea pachetelor are o influenţă decisivă asupra locaţiei unde trebuie să fie salvate fişierele sursă ale acestei aplicaţii de test, considerăm directorul de lucru `c:\javawork` (pentru utilizatorii de Windows). Analog, utilizatorii altor sisteme de operare, vor putea implementa exemplul ales, considerând un director de lucru specific platoformei respective (`/usr/local/javawork` este un exemplu pentru platforma Linux).

Digitized by Google

Câteva detalii despre pachete sunt necesare în acest moment al construirii aplicaţiei. În primul rând, numele pachetelor trebuie să fie unice, pentru a evita eventualele conflicte care pot să apară la încărcarea claselor. Existenţa a două pachete cu acelaşi nume pune maşina virtuală Java în situaţia de a nu şti ce clase să încarce, din primul pachet sau din al doilea. Consecinţa imediată este faptul că nu se pot crea pachete cu acelaşi nume ca al pachetelor predefinite de mediul de programare Java (de exemplu `java.lang`, `java.util` etc). Sarcina de a găsi un nume unic pentru fiecare pachet poate fi uşurată dacă sunt respectate convenţiile de programare Java (detalii în anexa B). Potrivit acestor convenţii, putem crea pachete pornind de la domeniul Internet deţinut, în cazul în care programatorul/firma deţine un astfel de nume (de exemplu pachetul `ro.firmamea` provine de la domeniul `www.firmamea.ro`). Dacă nu există un domeniu Internet, este necesară crearea unei combinaţii de nume, care este puţin probabil să se repete. Afirmaţiile anterioare sunt utile mai ales în cazul în care aplicaţiile create de dumneavoastră sunt redistribuite către diverşi utilizatori, care ar putea folosi la rândul lor alte pachete, de la alţi furnizori. Exemplul nostru fiind unul de test, am ales ca nume de pachet `com.test`, nume care poate asigura unicitatea la nivelului calculatorului pe care realizăm aplicaţiile Java de test, deşi nu este personalizat cu informaţii cu caracter unic (de exemplu, numele persoanei sau al firmei care l-a creat).

Numele pachetului presupune crearea unei ierarhii de directoare conformă cu el. În cazul nostru, cele trei pachete existente `com.test`, `com.test.masini` şi `com.test.posesori`, presupun crearea următoarelor directoare: `com`, `com\test`, `com\test\masini` şi `com\test\posesori`, conform ierarhiei:

```
-com
  |
   - test
       |
        - masini
       |
        - posesori
```

Cele patru directoare sunt create în cadrul directorului de lucru stabilit anterior (`c:\javawork`). În fiecare dintre aceste directoare se află fişierele sursă ale claselor precizate anterior, în funcţie de pachetul în care sunt definite:

- `Dacia.java`, `Ferrari.java` se găsesc în directorul `c:\javawork\com\test\masini`;

Digitized by Google

- `Persoana.java` şi `Firma.java` în directorul `c:\javawork\com\test\posesori;`
- `ExempluPachete.java` în directorul `c:\javawork\com\test`.

Codul sursă al acestor fişiere este prezentat în continuare:

Listing C.1: Codul sursă al clasei `Dacia`

```
//Fisierul com\test\masini\Dacia.java
package com.test.masini;

public class Dacia
{
    /** Modelul masinii (1310, 1300, Berlina, etc.). */
    private String tip;


    /** Creaza o Dacie cu modelul precizat. */
    public Dacia(String tipul)
    {
        tip = tipul;
    }

    /** Returneaza tipul masinii. */
    public String returneazaTip()
    {
        return tip;
    }
}
```

Listing C.2: Codul sursă al clasei `Ferrari`

```
//Fisierul com\test\masini\Ferrari.java
package com.test.masini;

public class Ferrari
{
    /** Modelul masinii (F50, etc.). */
    private String tip;


    /** Creaza un Ferrari cu modelul precizat. */
    public Ferrari(String tipul)
    {
        tip = tipul;
    }

    /** Returneaza tipul masinii. */
    public String returneazaTip()
```


Digitized by Google

```
    {
        return tip;
    }
}
```

Listing C.3: Codul sursă al clasei `Persoana`

```
// Fisierul com\test\posesori\Persoana.java
package com.test.posesori;

import com.test.masini.*;
import java.util.*;

public class Persoana
{
    /** Numele persoanei.*/
    private String nume;

    /** Adresa persoanei.*/
    private String adresa;

    /** Sir cu masinile Dacia detinute de persoana*/
    private Vector masini;


    /** Creaza o persoana cu numele si adresa specificate.*/
    public Persoana(String numele, String adresa_)
    {
        nume = numele;
        adresa = adresa_;
        masini = new Vector();
    }

    /** Returneaza nume.*/
    public String returneazaNume()
    {
        return nume;
    }

    /** Adauga o masina Dacia celor aflate deja in posesie.*/
    public void adaugaDacia(Dacia masina)
    {
        masini.add(masina);
    }

    /** Returneaza masinile aflate in posesie.*/
    public Vector returneazaMasini()
    {
        return masini;
    }
```


Digitized by Google

```
}
```

Listing C.4: Codul sursă al clasei `Firma`

```
//Fisierul com\test\posesori\Firma.java
package com.test.posesori;

import java.util.*;
import com.test.masini.*;

public class Firma
{
    /** Numele firmei.*/
    private String nume;

    /** Numar angajati.*/
    private int numarAngajati;

    /** Sir cu masinile aflate in posesie.*/
    private Vector masini;


    /**
     * Creaza o firma specificand numele si
     * numarul de angajati.
     */
    public Firma(String numele, int numarulAngajatilor)
    {
        nume = numele;
        numarAngajati = numarulAngajatilor;
        masini = new Vector();
    }

    /** Adauga un Ferrari celor aflate deja in posesie.*/
    public void adaugaFerrari(Ferrari masina)
    {
        masini.add(masina);
    }

    /** Returneaza nume.*/
    public String returneazaNume()
    {
        return nume;
    }

    /** Returneaza masinile aflate in posesie.*/
    public Vector returneazaMasini()
    {
        return masini;
    }
```


Digitized by Google

```
}
```

Listing C.5: Utilizarea claselor din pachetele definite

```java
//Fisierul com\test\ExempluPachete.java
package com.test;

import com.test.masini.*;
import com.test.posesori.*;
import java.util.*;

public class ExempluPachete
{
    public static void main(String[] args)
    {
        Persoana p = new Persoana("Mircea Ionescu", "");
        Dacia d = new Dacia("1300");

        //persoana P detine masina d
        p.adaugaDacia(d);
        Vector v = p.returneazaMasini();

        //afisarea tipurilor masinilor detinute de persoana
        System.out.println("Tipurile de masini detinute de "
                + p.returneazaNume() + ": ");
        for(int i = 0; i < v.size(); i++)
        {
            String tip = ((Dacia) v.get(i)).returneazaTip();
            System.out.print(tip + " ");
        }
        System.out.println("");

        Firma firmaMea = new Firma("Sun", 13000);
        Ferrari f = new Ferrari("F50");

        //firma firmaMea detine masina f
        firmaMea.adaugaFerrari(f);
        v = firmaMea.returneazaMasini();

        System.out.println("Tipurile de masini detinute de "
                + firmaMea.returneazaNume() + ": ");
        for(int i = 0; i < v.size(); i++)
        {
            String tip = ((Ferrari) v.get(i)).returneazaTip();
            System.out.print(tip + " ");
        }
    }
}
```

După ce clasele au fost create, următoarea acţiune constă în setarea vari-

Digitized by Google

abilei de sistem CLASSPATH, prezentată în cadrul capitolului 4.4.3. Variabila CLASSPATH trebuie astfel setată încât să conţină şi directorul de lucru pentru aplicaţia noastră de test. Pentru aceasta, utilizatorii de Windows execută următoarea comandă în cadrul unui prompt MS-DOS:

```
set CLASSPATH=%CLASSPATH%;c:\javawork\
```

Pentru a verifica dacă variabila CLASSPATH a fost actualizată, se lansează în acelaşi prompt MS-DOS comanda:

```
echo %CLASSPATH%
```

Comanda afişează informaţii asemănătoare cu cele ce urmează:

```
.;c:\jdk\lib\tools.jar;c:\jdk\jre\lib\rt.jar;
         c:\javawork\
```

Dacă printre locaţiile afişate se află directorul de lucru, atunci se poate trece la pasul următor.

Următorul pas constă în compilarea claselor definite în cadrul celor trei pachete care compun exemplul. Pentru ca procesul de compilare să fie mai uşor, se crează un fişier cu numele fisiereSursa, în directorul de lucru (c:\javawork), având următorul conţinut:

```
com\test\ExempluPachet.java
com\test\masini\Dacia.java
com\test\masini\Ferrari.java
com\test\posesori\Persoana.java
com\test\posesori\Firma.java
```

Apoi, în acelaşi prompt MS-DOS în care s-a setat variabila CLASSPATH, se compilează fişierele sursă ale aplicaţiei test, utilizând din directorul de lucru comanda:

```
javac @fisiereSursa
```

Este foarte important ca toate comenzile să fie executate în acelaşi prompt MS-DOS în care s-a setat variabila CLASSPATH, pentru că această setare este valabilă doar în cadrul acelui prompt. În momentul în care promptul respectiv este închis, se pierde modificarea realizată asupra variabilei CLASSPATH, revenindu-se la valoarea iniţială. Pentru a face aceste modificări permanente, ele trebuie să fie efectuate în:

Digitized by Google

- fişierul `autoexec.bat` din directorul rădăcină, în cazul sistemelor Windows 95, 98, ME, urmate de repornirea calculatorului;
- secţiunea `User Variables` din `Control Panel-> System-> Environment`, în cazul sistemelor Windows NT, 2000.

Deoarece numărul de fişiere sursă este destul de mic pentru acest exemplu, folosirea comenzii anterioare este acceptabilă. Dar, dacă aplicaţia pe care o realizaţi are un număr mare de clase, atunci este recomandat să utilizaţi utilitarul `ant`, prezentat în anexa A, pentru a compila fişierele aflate într-o ierarhie de directoare.

Prin compilare, se crează fişierele `.class` respective, în cadrul aceloraşi directoare cu cele sursă. Dacă se doreşte separarea fişierelor sursă de cele `.class`, atunci comanda `javac` trebuie folosită cu opţiunea `-d`, urmată de directorul unde vor fi create fişierele `.class`. Trebuie menţionat că acest nou director nu este automat precizat în variabila `CLASSPATH`. De aceea, dacă optaţi pentru această variantă, va trebui să adăugaţi personal directorul respectiv în cadrul variabilei `CLASSPATH`.

Exemplul construit în această secţiune se execută prin rularea în directorul de lucru a comenzii:

```
java com.test.ExempluPachet
```

Rezultatele afişate sunt:

```
Tipurile de masini Dacia detinute de Mircea Ionescu:
1300
Tipurile de masini Ferrari detinute de Sun:
F50
```

Pentru a înţelege în profunzime modul de lucru cu pachete, să vedem cum găseşte Java pachetele şi clasele definite de programator. Mai întâi, interpretorul Java (`java`) caută variabila de sistem `CLASSPATH`. Pornind de la rădăcină, interpretorul preia, pe rând, fiecare nume de pachet şi înlocuieşte fiecare simbol `"."` întâlnit cu simbolul `"\"` sau `"/"` în funcţie de sistemul de operare pe care rulează, pentru a genera o cale relativă unde vor fi căutate fişierele `.class`. În cazul nostru, pachetul `com.test.masini` este transformat în `com\test\masini`. Apoi, această cale relativă este concatenată la diferitele intrări din variabila de sistem `CLASSPATH`, de exemplu `c:\javawork` sau `c:\jdk\lib\tools.jar` etc. Fiecare dintre aceste concatenări nu reprezintă altceva decât căi absolute, unde interpretorul caută fişierele `.class` (adică

Digitized by Google

Dacia.class, Ferrari.class). Pentru exemplul nostru, interpretorul găseşte fişierele anterioare în directorul c:\javawork\com\test\masini.

În acest moment se poate spune că primele pachete realizate de dumneavoastră sunt funcţionale. Exemplul ales a fost însă unul simplu pentru a putea înţelege paşii necesari de la crearea pachetelor şi până la rularea aplicaţiei care le foloseşte. Cu siguranţă că de-a lungul timpului nu vă veţi opri aici şi acest exemplu va fi urmat de altele, conţinând ierarhii de pachete din ce în ce mai complexe.

C.3 Arhive jar (Java ARchives)

Arhivele jar permit gruparea mai multor fişiere într-un sigur fişier arhivat. În mod obişnuit fişierele conţinute de o arhivă jar sunt fişiere .class, împreună cu diverse resurse care pot fi asociate cu o aplicaţie (imagini, etc.). Nu există o interdicţie în ceea ce priveşte tipul de fişiere conţinute de o arhivă jar (acestea pot conţine orice tip de fişiere, ca de altfel orice alt format de arhivare, cum ar fi zip, rar, ace, arj etc.), dar în general sunt folosite fişierele .class.

Arhivele jar oferă programatorilor anumite beneficii:

- *compresie*: arhivele jar permit compresarea fişierelor pentru a fi stocate mai eficient;

- *simplificarea distribuirii aplicaţiilor*: aplicaţia creată este distribuită clienţilor sub forma unui singur fişier jar, care înglobează toate fişierele şi resursele necesare funcţionării aplicaţiei;

- *micşorarea timpului de transfer*: pentru a fi descărcate fişierele conţinute de un fişier jar, este nevoie doar de o singură conexiune HTTP şi nu de câte o conexiune pentru fiecare fişier în parte;

- *portabilitate*: ca parte a platformei Java, arhivele jar sunt portabile, putând fi utilizate fără a fi modificate pe orice platformă cu suport Java;

- *extinderea funcţionalităţii platformei Java*: mecanismul de extindere oferit de platforma Java permite extinderea funcţionalităţii platformei prin adăugarea unor fişiere jar celor existente (printre arhivele existente în orice implementare Java se numără şi rt.jar, ce conţine API-ul Java predefinit, adică toată ierarhia de clase pe care mediul Java o oferă spre utilizare programatorilor). Prin adăugarea unor arhive noi, puteţi să extindeţi funcţionalitatea platformei Java. Un exemplu este extensia *JavaMail*,

Digitized by Google

dezvoltată de Sun, care reprezintă un API utilizat în programele Java pentru transmiterea şi recepţionarea de email-uri;

- *versionarea pachetelor de clase*: un fişier `jar` poate cuprinde date despre fişierele pe care le conţine, cum ar fi informaţii despre cel care a creat pachetele şi despre versiunea pachetelor respective.

C.3.1 Utilizarea fişierelor `.jar`

Fişierele `jar` sunt arhivate pe baza formatului `zip`, deci au aceleaşi facilităţi ca orice alt tip de arhivă (compresie fără pierderea datelor etc.). Pe lângă aceste facilităţi de bază, fişierele `jar` au şi alte funcţionalităţi mai avansate (rularea unei aplicaţii în format `jar` este una dintre ele), a căror înţelegere este condiţionată de familiarizarea cu operaţiile fundamentale cu arhive `jar`: crearea unei arhive `jar`, vizualizarea, extragerea sau modificarea conţinutului acesteia.

Operaţiile de bază cu arhivele `jar` se realizează folosind o aplicaţie utilitară oferită de platforma Java. Această aplicaţie utilitară este în linie de comandă şi este denumită `jar`, putând fi găsită în directorul `bin` al distribuţiei Java instalată pe calculatorul dumneavoastră. Pentru a testa prezenţa utilitarului pe calculatorul dumneavoastră, deschideţi o fereastră de comandă (sau, altfel spus, un *prompt*) şi rulaţi comanda `jar`. Implicit, vor fi afişate informaţii ajutătoare despre această comandă, cum ar fi opţiunile de utilizare şi câteva exemple.

Iată, pentru început, un rezumat al celor mai frecvente operaţii cu arhive `jar`:

- crearea unei arhive jar (cu alte cuvinte, "împachetarea" fişierelor şi a directoarelor într-un singur fişier);
- vizualizarea conţinutului unei arhive `jar;`
- extragerea conţinutului unei arhive `jar;`
- extragerea doar a anumitor fişiere conţinute într-o arhivă `jar;`
- modificarea conţinutului unei arhive `jar;`
- rularea unei aplicaţii "împachetate" într-o arhivă `jar;`

C.3.2 Crearea unui fişier `.jar`

O arhivă `jar` este realizată utilizând aplicaţia utilitară `jar` în următoarea formă:

Digitized by Google

```
jar cf nume_fisier_jar nume_fisiere
```

Opţiunile şi argumentele utilizate în această comandă au următoarea semnificaţie:

- opţiunea `c` indică faptul că este *creată* o arhivă `jar`;
- opţiunea `f` indică faptul că arhiva este creată într-un fişier. Este important de reţinut faptul că opţiunile nu trebuie separate prin spaţiu şi că ordinea de apariţie a lor nu contează;
- argumentul `nume_fisier_jar` reprezintă numele fişierului `jar` rezultat în urma arhivării. Poate fi folosit orice nume pentru a denumi un fişier `jar`. Prin convenţie, fişierele au extensia `.jar`, deşi acest lucru nu este obligatoriu;
- argumentul `nume_fisiere` reprezintă o listă de nume de fişiere şi directoare, separate prin spaţiu, care vor fi introduse în arhivă. Numele acestora poate să conţină şi simbolul `*`, pentru a face referire la mai multe fişiere în acelaşi timp. În cazul directoarelor, conţinutul acestora este adăugat recursiv la arhivă;

Suplimentar, se pot folosi şi alte opţiuni, pe lângă cele de bază (`cf`):

- opţiunea `v` afişează numele fiecărui fişier adăugat la arhivă, în timpul creării acesteia;
- opţiunea `0` indică faptul că arhiva nu va fi compresată;
- opţiunea `M` indică faptul că fişierul manifest asociat arhivei nu este creat (detalii despre fişierele manifest sunt oferite mai târziu în cadrul secţiunii curente);
- opţiunea `m` indică faptul că pot fi preluate dintr–un fişier manifest existent informaţii necesare arhivei care va fi creată. În acest caz, comanda are următoarea formă:

  ```
  jar cfm nume_fisier_manifest nume_fisier_jar
              nume_fisiere
  ```

- opţiunea `-C` schimbă directorul curent în timpul execuţiei comenzii.

Digitized by Google

În continuare vom prezenta câteva exemple de creare a arhivelor `jar`, care utilizează fişierele `.class` ale aplicaţiei de test cu maşini şi posesori, prezentată în subcapitolul precedent. Aceste fişiere se află conform celor stabilite în subcapitolul precedent, în directorul `c:\javawork`, care are următoarea structură (pentru simplitate, sugerăm mutarea celorlalte tipuri de fişiere în alt director):

```
- com
    - test
        - masini
            - Dacia.class
            - Ferrari.class
        - posesori
            - Firma.class
            - Persoana.class
        - ExempluPachete.class
```

În primul exemplu de creare a arhivelor `jar`, în directorul `c:\javawork` se lansează dintr–o fereastră de comandă aplicaţia `jar` astfel:

```
jar cvf pachete.jar com
```

În fereastra de comandă se va vedea ceva asemănător cu textul următor:

```
added manifest
adding: com/(in = 0) (out= 0)(stored 0%)
adding: com/test/(in = 0) (out= 0)(stored 0%)
adding: com/test/masini/(in = 0) (out= 0)(stored 0%)
adding: com/test/masini/Dacia.class(in = 364)
(out= 257) (deflated 29%)
adding: com/test/masini/Ferrari.class(in = 368)
(out= 259) (deflated 29%)
adding: com/test/posesori/(in = 0) (out= 0)(stored 0%)
adding: com/test/posesori/Persoana.class(in = 738)
(out= 432) (deflated 41%)
adding: com/test/posesori/Firma.class(in = 730)
(out= 439) (deflated 39%)
adding: com/test/ExempluPachete.class(in = 1602)
(out= 913) (deflated 43%)
```

Deoarece `com` este un director, utilitarul `jar` a adăugat recursiv toate directoarele şi fişierele conţinute în el. Rezultatul execuţiei acestei comenzi este

Digitized by Google

fişierul `pachete.jar`, aflat în directorul curent, în care a fost lansată comanda. Analizând textul afişat se poate observa că fişierul proaspăt creat este comprimat.

Există şi posibilitatea de a crea o arhivă `pachete.jar` necomprimată. Pentru aceasta se lansează comanda:

```
jar cvf0 pachete1.jar com
```

Aceasta afişează următorul text:

```
added manifest
adding: com/(in = 0) (out= 0)(stored 0%)
adding: com/test/(in = 0) (out= 0)(stored 0%)
adding: com/test/masini/(in = 0) (out= 0)(stored 0%)
adding: com/test/masini/Dacia.class(in = 364)
(out= 364) (stored 0%)
adding: com/test/masini/Ferrari.class(in = 368)
(out= 368) (stored 0%)
adding: com/test/posesori/(in = 0) (out= 0)(stored 0%)
adding: com/test/posesori/Persoana.class(in = 738)
(out= 738) (stored 0%)
adding: com/test/posesori/Firma.class(in = 730)
(out= 730) (stored 0%)
adding: com/test/ExempluPachete.class(in = 1602)
(out= 1602) (stored 0%)
```

Am ales să dăm o altă denumire arhivei din al doilea exemplu, pentru a putea compara dimensiunile celor două fişiere `jar` rezultate. Dacă cel comprimat, denumit `pachete.jar`, are `3.792` de `bytes`, cel necomprimat, `pachete1.jar`, are `5.180` de `bytes`.

Puteţi să creaţi o arhivă `jar` folosind simbolul `*`, ca în exemplul următor:

```
jar cvf pachete.jar *
```

Cu ajutorul acestui simbol sunt adăugate în arhivă toate fişierele şi (recursiv) directoarele din directorul curent.

Ultimul exemplu pe care îl prezentăm foloseşte opţiunea `-C`, de schimbare a directorului curent. Prin utilizarea acestei opţiuni, căile relative nu mai sunt păstrate în totalitate în cadrul fişierului `jar`. Pentru a observa acest comportament al utilitarului `jar`, lansaţi comanda:

```
jar cvf pachete.jar -C com\test\ .
```


Digitized by Google

În fereastra de comandă se va afişa:

```
added manifest
adding: masini/(in = 0) (out= 0)(stored 0%)
adding: masini/Dacia.class(in = 364) (out= 257)
(deflated 29%)
adding: masini/Ferrari.class(in = 368) (out= 259)
(deflated 29%)
adding: posesori/(in = 0) (out= 0)(stored 0%)
adding: posesori/Persoana.class(in = 738) (out= 432)
(deflated 41%)
adding: posesori/Firma.class(in = 730) (out= 439)
(deflated 39%)
adding: ExempluPachete.class(in = 1602) (out= 913)
(deflated 43%)
```

Cu alte cuvinte, în timpul execuţiei comenzii, directorul curent a fost schimbat în `com\test`, iar fişierele şi directoarele au fost adăugate ca şi cum comanda ar fi fost executată în acel director. Utilitatea acestei opţiuni apare în momentul în care se adaugă mai multe fişiere şi directoare la o arhivă, deoarece atunci se poate schimba dinamic, dintr-o singură comandă, directorul curent de execuţie a utilitarului `jar`.

C.3.3 Vizualizarea conţinutului unui fişier `.jar`

Conţinutul unui fişier `jar` poate fi vizualizat fără a-l extrage propriu-zis din interiorul fişierului. Comanda standard utilizată pentru a realiza acest lucru este:

```
jar tf nume_fisier_jar
```

Opţiunile şi argumentele utilizate de această comandă au următoarea semnificaţie:

- opţiunea `t` indică faptul că se doreşte vizualizarea conţinutului unui fişier `jar`;
- opţiunea `f` indică faptul că în linia de comandă va fi specificat un nume de fişier, al cărui conţinut va fi afişat;
- argumentul `nume_fisier_jar` reprezintă numele fişierului `jar` al cărui conţinut va fi afişat.

Digitized by Google

Suplimentar, se poate utiliza şi opţiunea v pentru a obţine informaţii despre dimensiunea fişierelor care intră în componenţa arhivei jar şi data în momentul ultimei modificări la care au fost supuse.

Ca exemplu, lansaţi în directorul c:\javawork următoarea comandă pentru a vedea care este conţinutul arhivei pachete.jar, creată la secţiunea anterioară (primul exemplu din suită):

```
jar tf pachete.jar
```

Rezultatul execuţiei acestei comenzi este următorul:

```
META-INF/
META-INF/MANIFEST.MF
com/
com/test/
com/test/masini/
com/test/masini/Dacia.class
com/test/masini/Ferrari.class
com/test/posesori/
com/test/posesori/Persoana.class
com/test/posesori/Firma.class
com/test/ExempluPachete.class
```

După cum se poate observa, este afişat fiecare director şi fişier conţinut în fişierul jar. Directorul META-INF şi fişierul MANIFEST.MF, conţinut în acest director, reprezintă o componentă specială a oricărui fişier jar (paragraful C.3.5), fiind plasate în fişier la momentul creării acestuia. De asemenea, este util de reţinut că toate directoarele şi fişierele listate au căi relative, de tipul celei de mai jos: com/test/masini/Dacia.class.

C.3.4 Extragerea conţinutului unui fişier .jar

Forma standard a comenzii cu ajutorul căreia se extrage conţinutul unui fişier jar este următoarea:

```
jar xf nume_fisier_jar [nume_fisiere_arhivate]
```

Opţiunile şi argumentele utilizate au semnificaţia:

- opţiunea x indică faptul că este *extras* conţinutul unui fişier jar;
- opţiunea f specifică faptul că fişierul jar al cărui conţinut va fi extras, este specificat în linie de comandă;

Digitized by Google

- argumentul `nume_fisier_jar` reprezintă numele fişierului al cărui conţinut va fi extras;
- argumentul `nume_fisiere_arhivate` este un argument opţional (motiv pentru care apare între caracterele `[ ]`), reprezentând lista cu numele fişierelor care vor fi extrase din fişierul `jar`, separate prin spaţiu. Dacă argumentul lipseşte, atunci va fi extras întreg conţinutul fişierului `jar`.

Când extrage conţinutul unui fişier `jar`, utilitarul `jar` creează copii ale fişierelor şi directoarelor care sunt extrase, reproducând structura de directoare pe care fişierele o au în cadrul arhivei. Toate aceste copii sunt create în directorul curent, fişierul `jar` rămânând intact. Însă, dacă există fişiere cu acelaşi nume în directorul curent, ele vor fi rescrise.

Ca exemplu, să extragem conţinutul arhivei `pachete.jar`. Pentru aceasta, executaţi comanda următoare din directorul `c:\javawork`, într-o fereastră de comandă:

```
jar xf pachete.jar
```

Rezultatul execuţiei acestei comenzi sunt cele două directoare conţinute în arhivă, împreună cu subdirectoarele şi fişierele pe care le conţin. Aceste directoare sunt `com`, unde se află clasele aplicaţiei de test, şi `META-INF`, unde se găseşte fişierul `MANIFEST.MF`. După cum se observă, toate fişierele şi directoarele conţinute au fost extrase, iar fişierul `pachete.jar` a rămas intact.

Pentru a extrage doar anumite fişiere din arhiva `jar`, trebuie să le specificaţi ca argument. Ca exemplu, executaţi comanda:

```
jar xf pachete.jar com/test/posesori/Firma.class
```

Această comandă creează ierarhia de directoare `com/test/posesori`, în cazul în care nu există în directorul curent, şi apoi creează o copie a fişierului `Firma.class` în directorul `posesori`.

C.3.5 Fişierul manifest al unei arhive `jar`

Fişierul manifest este un fişier special al unei arhive `jar`, care conţine informaţii despre fişierele arhivate. Implicit, fişierul are numele `MANIFEST.MF`.

Crearea unei arhive `jar` presupune, implicit, crearea unui fişier manifest, asociat cu respectiva arhivă. Fiecare arhivă `jar` are un singur fişier manifest, care este automat creat în directorul `META-INF` al arhivei. Numele fişierului manifest şi al directorului în care se află acesta, nu pot fi modificate, deci întotdeauna fişierul manifest are locaţia `META-INF/MANIFEST.MF`.

Digitized by Google

Informaţiile conţinute într-un fişier manifest sunt de tipul "cheie: valoare". De exemplu, fişierul manifest al arhivei `pachete.jar` are următorul conţinut:

```
Manifest-Version: 1.0
Created-By: 1.4.0 (Sun Microsystems Inc.)
```

Fişierul manifest conţine două informaţii despre arhiva `pachete.jar`: versiunea fişierului manifest (specificată de cheia `Manifest-Version`) şi numele platformei Java care a creat fişierul manifest respectiv (specificat de cheia `Created-By`).

Acesta este formatul standard al unui fişier manifest asociat unei arhive `jar`. Informaţiile pe care le oferă diferă însă de la o arhivă la alta, în funcţie de scopul arhivei. Cu alte cuvinte, pe lângă acestea mai pot apare şi altele, care vor fi prezentate în continuare. Dacă rolul pe care îl are arhiva `jar` este doar cel standard al unei arhive, cum ar fi compresia datelor, nu trebuie să vă faceţi probleme în legătură cu fişierul manifest, pentru că fişierul manifest nu are nici un rol în astfel de situaţii. Dacă, însă, doriţi ca arhiva `jar` să prezinte funcţionalităţi speciale (avansate), atunci va trebui să modificaţi conţinutul fişierului manifest, dar pentru aceasta mai întâi va trebui să cunoaşteţi *ce* trebuie modificat în fişierul manifest şi, după aceea, *cum* trebuie să-l modificaţi.

Pentru ca o aplicaţie "împachetată" într-un fişier `jar`, să poată fi executată folosind acest format, este necesară specificarea "punctului de intrare" al aplicaţiei, adică numele clasei în care se află metoda `main()`, care este apelată în cazul în care aplicaţia nu este arhivată într-un fişier `jar`.

Specificarea numelui clasei respective se realizează utilizând cheia `Main--Class`. Pentru aplicaţia noastră de test, informaţia arată astfel:

```
Main-Class: com.test.ExempluPachete
```

Dacă aplicaţia dumneavoastră utilizează clase disponibile în alte arhive `jar`, atunci trebuie specificat acest lucru prin intermediul cheii `Class-Path`. Deşi în cazul aplicaţiei noastre nu se folosesc clase din alte arhive `jar`, iată un exemplu de prezentare a acestei informaţii:

```
Class-Path: test2.jar app/test3.jar
```

Pentru a reţine informaţii referitoare la versiunea pachetelor conţinute într-o arhivă `jar`, se pot utiliza chei speciale. Iată un exemplu, adaptat la datele existente în aplicaţia noastră:

```
Name: com/test/masini
```


Digitized by Google

```
Specification-Title: "Clase de masini"
Specification-Version: "1.0"
Specification-Vendor: "X Software"
Implementation-Title: "com.test.masini"
Implementation-Version: "versiune de test 0.12"
Implementation-Vendor: "X Software"
```

Un astfel de set de informaţii poate fi prezentat pentru fiecare pachet al aplicaţiei.

C.3.6 Modificarea fişierului manifest al unei arhive `jar`

Modificarea fişierului manifest al unei arhive `jar` se realizează prin intermediul opţiunii `m` pe care o oferă utilitarul `jar`. Opţiunea `m` permite adăgarea în fişierul manifest a informaţiilor dorite, în timpul operaţiei de creare a arhivei `jar`.

Formatul standard al comenzii este:

```
jar cfm nume_fisier_manifest nume_fisier_jar
            nume_fisiere
```

Dat fiind faptul că prin această comandă se crează un fişier `jar`, opţiunile `cf` şi argumentele `nume_fisier_jar` şi `nume_fisiere` au aceeaşi semnificaţie cu cea descrisă la secţiunea de creare a unui fişier `jar`. Prin urmare, singurele lucruri care mai trebuie precizate sunt:

- opţiunea `m` specifică faptul că se vor adăuga anumite informaţii în fişierul manifest creat automat la crearea unei arhive `jar`;
- argumentul `nume_fisier_manifest` reprezintă numele fişierului din care vor fi preluate informaţiile care vor fi adăugate în fişierul manifest.

Ca exemplu, vom recrea arhiva `pachete.jar`, adăugând fişierului manifest câteva informaţii cu caracter special. Astfel, în directorul curent, `c:\javawork`, se crează un fişier cu numele `info.txt`, având următorul conţinut:

```
Main-Class: com.test.ExempluPachete

Name: com/test/masini
Specification-Title: "Clase de masini"
Specification-Version: "1.0"
Specification-Vendor: "CompaniaMea"
```


Digitized by Google

```
Implementation-Title: "com.test.masini"
Implementation-Version: "versiune de test 0.12"
Implementation-Vendor: "X Software"
```

După ce fişierul este salvat, se execută comanda:

```
jar cmf info.txt pachete.jar com
```

Rezultatul execuţiei acestei comenzi este arhiva `pachete.jar`. Pentru a vedea modificările efectuate în fişierul manifest, extrageţi acest fişier din arhivă. Fişierul ar trebui să arate astfel:

```
Manifest-Version: 1.0
Main-Class: com.test.ExempluPachete
Created-By: 1.4.0 (Sun Microsystems Inc.)

Name: com/test/masini
Specification-Title: "Clase de masini"
Specification-Vendor: "X Software"
Implementation-Vendor: "X Software"
Specification-Version: "1.0"
Implementation-Version: "versiune de test 0.12"
Implementation-Title: "com.test.masini"
```

Se observă că noul fişier manifest este o combinaţie între fişierul standard creat de utilitarul `jar` şi fişierul `info.txt`.

C.3.7 Modificarea conţinutului unui fişier `.jar`

Conţinutul unui fişier `jar` poate fi modificat folosind opţiunea `u` a utilitarului `jar`. Prin modificarea conţinutului unui fişier `jar` se poate înţelege atât modificarea fişierului manifest asociat arhivei `jar`, cât şi adăugarea unor noi fişiere în arhiva `jar`.

Forma standard a comenzii este:

```
jar uf nume_fisier_jar nume_fisiere
```

Opţiunile şi argumentele utilizate au următoarea semnificaţie:

- opţiunea `u` semnifică faptul că arhiva `jar` va fi modificată;
- opţiunea `f` specifică faptul că numele fişierului care va fi modificat este precizat în linie de comandă;

Digitized by Google

- argumentul `nume_fisier_jar` reprezintă numele fişierului care este modificat (actualizat);

- argumentul `nume_fisiere` reprezintă o listă de fişiere ce vor fi adăugate în arhivă.

Dacă se utilizează opţiunea `m` (asemănător cu secţiunea anterioară, de modificare a fişierului manifest), atunci se poate modifica fişierul manifest al arhivei `jar`, concomitent cu adăugarea unor noi fişiere în arhiva `jar`.

Dacă, de exemplu, se doreşte adăugarea în arhiva `pachete.jar` a unui nou fişier `new.gif`, aflat în directorul curent, atunci se execută comanda:

```
jar uf pachete.jar new.gif
```

De asemenea, se poate modifica fişierul manifest al arhivei `pachete.jar`, prin execuţia următoarei comenzi (fişierul `info.txt` este cel definit la secţiunea anterioară):

```
jar umf info.txt pachete.jar
```

C.3.8 Rularea aplicaţiilor Java "împachetate" într-o arhivă `jar`

Aplicaţiile care sunt arhivate în cadrul unui fişier `jar` pot fi rulate folosind comanda:

```
java -jar nume_fisier_jar
```

Opţiunea `-jar` specifică faptul că aplicaţia ce va fi rulată este conţinută într-o arhivă `jar`. Această comandă poate fi utilizată doar dacă în fişierul manifest al arhivei `jar` este specificat "punctul de intrare" al aplicaţiei, prin intermediul cheii `Main-Class`. În consecinţă, pentru a putea rula o aplicaţie conţinută într-o arhivă `jar`, trebuie mai întâi să modificaţi fişierul manifest pentru a introduce informaţia cerută.

În exemplul nostru, odată ce valoarea cheii `Main-Class` este specificată, se poate rula aplicaţia utilizând comanda:

```
java -jar pachete.jar
```

Rezultatul execuţiei acestei comenzi este următorul:

Digitized by Google

```
Tipurile de masini Dacia detinute de Mircea Ionescu:
1300
Tipurile de masini Ferrari detinute de Sun:
F50
```

În concluzie, secţiunea destinată arhivelor `jar` prezintă modalitatea de a realiza operaţii de bază cu arhive `jar`, precum şi modalitatea de a rula aplicaţii conţinute în cadrul unei astfel de arhive, înfăţişând aceste caracteristici într-un mod simplu, la obiect, plin de exemple elocvente, pentru ca programatorul să poată surprinde facilităţile puternice pe care le oferă aceste arhive specifice platformei Java.

Digitized by Google

D. Internaţionalizarea aplicaţiilor. Colecţii de resurse

> Pentru un bărbat curajos, fiecare ţară reprezintă o patrie.
>
> Proverb grecesc

Lumea în care trăim este mică şi tinde să devină din ce în ce mai mică pe zi ce trece. În mare măsură, acesta este rezultatul implicării computerelor în viaţa noastră. Internetul a devenit în ultimii ani principalul mijloc de comunicare între oameni, pentru că oferă o modalitate foarte simplă şi rapidă de comunicare. Emailul sau navigarea pe Internet nu mai reprezintă o necunoscută pentru majoritatea dintre noi. Practic, putem spune că trăim într-un imens sat, în care fiecare poate comunica simplu şi rapid cu ceilalţi. Efectul globalizării se simte şi în industria IT, unde programatorii trebuie să considere ca piaţă de desfacere întreaga planetă. Ei sunt nevoiţi să dezvolte aplicaţii care pot fi utilizate din Los Angeles până la Paris, fără ca acest lucru să implice un efort prea mare de programare. Cu alte cuvinte, sarcina lor este de a crea aplicaţii internaţionalizate. Din acest punct de vedere, programatorii Java pot să fie liniştiţi. J2SDK oferă o întreagă ierarhie de clase care permite adaptarea unei aplicaţii de la o limbă/cultură la alta, cu un efort minim de implementare. În Java, procesul este chiar mai simplu decât pare.

D.1 Timpul, numerele şi datele calendaristice

Este foarte bine cunoscut faptul că, datorită diverselor popoare şi culturi existente pe planeta noastră, nu există un standard acceptat la scară planetară pentru afişarea numerelor, a datelor calendaristice sau a timpului.

Digitized by Google

Să luăm ca exemplu următoarea dată: `5/3/02`. Pentru cei ce locuiesc în Statele Unite, această dată reprezintă data de 3 mai 2002, în timp ce pentru cei ce locuiesc în Europa, aceasta reprezintă 5 martie 2002. După cum se poate observa, modul în care componentele unei date (zi, lună, an) sunt combinate şi diferă de la o ţară la alta. În Statele Unite, formatul este în stilul lună/zi/an, în timp ce în majoritatea ţărilor din Europa formatul este zi/lună/an. Poate că nu este o diferenţă mare, dar nu trebuie subestimată importanţa ei pentru utilizatorii aplicaţiilor.

Formatele de date sunt poate cele mai simple. Ce se poate spune despre formatele de reprezentare a timpului sau cele de reprezentare a sumelor de bani (formate monetare) sau a numerelor? Răspunsul va fi descoperit pe parcursul acestei secţiuni a anexei.

Convenţiile de formate variază de la o ţară la alta, uneori semnificativ. Există situaţii în care aceste convenţii variază chiar şi în interiorul aceleaşi ţări. Iată câteva exemple:

- Timpul

  Afişarea orelor, minutelor şi secundelor pare să fie cea mai simplă dintre sarcini, dar nu este aşa. Diferenţe apar de exemplu la caracterul de separare a orelor de minute. Unele ţări folosesc simbolul `":"` (`11:13 PM`), în timp ce altele (Italia etc.) folosesc simbolul `"."` (`23.13`). De asemenea, cele 24 de ore ale zilei pot fi reprezentate ca jumătăţi de zi de câte 12 ore (primul exemplu), sau cu ore cuprinse între 0 şi 24 (al doilea exemplu);

- Datele calendaristice

  Exemplele anterioare au arătat că poziţia zilei, a lunii şi a anului într–o reprezentare de dată calendaristică diferă de la o zonă la alta. Pe de altă parte, caracterul de separare diferă şi el de la `"/"` în Statele Unite (de exemplu, `5/3/02`), la `"."` în Germania (de exemplu, `03.05.02`). De asemenea, numele zilelor şi ale lunilor diferă de la o limbă la alta;

- Numerele

  Primul lucru care atrage atenţia în cazul numerelor este separatorul zecimal. Statele Unite folosesc simbolul `"."` (de exemplu, `3.55`), dar alte ţări (România etc.) preferă simbolul `","` (de exemplu `3,55`). Pentru a îmbunătăţi citirea numerelor mai mari, cifrele sunt grupate de obicei, obţinându-se mii, milioane, miliarde etc. În Statele Unite acest simbol este `","` (de exemplu, `3,403.45`), iar în alte ţări (România etc.) acest simbol este `"."` (de exemplu, `3.403,45`). Sumele de bani sunt

Digitized by Google

reprezentate de asemenea diferit. Cei mai mulţi sunt familiarizaţi cu semnul `$` folosit de Statele Unite şi alte ţări (de exemplu, `$3,400.00`), dar există numeroase alte alternative.

De la lansarea ei, Java a fost catalogată drept soluţia dezvoltării de aplicaţii internaţionalizate. Dar până la apariţia JDK1.1, suportul pentru internaţionalizare a fost redus. Începând însă cu versiunea anterior amintită, Sun a inclus în distribuţia JDK pachetul `java.text` care conţine clasele şi interfeţele pentru manevrarea textului în formate locale specifice diverselor culturi.

Înainte de a prezenta soluţia Java pentru tratarea situaţiilor anterioare, este cazul să prezentăm noţiunea de localizare (*engl.* locale) şi clasa Java asociată, `Locale`.

O localizare poate desemna o regiune culturală, politică sau geografică. De exemplu, poate fi considerată regiunea din care provine un grup de persoane care vorbesc aceeaşi limbă. Din acest motiv, limba vorbită de aceste persoane este importantă pentru o localizare. Naţionalitatea de asemenea, pentru că, de exemplu, deşi americanii şi britanicii vorbesc aceeaşi limbă, ei au totuşi identităţi culturale diferite.

În Java, localizarea este reprezentată printr-o instanţă a clasei `Locale` din pachetul `java.util`. O instanţă a clasei `Locale` poate fi creată pentru orice limbă şi orice ţară. Localizarea se crează pe baza numelor limbii şi a ţării pentru care se realizează respectiva localizare. Ambele nume reprezintă de fapt abrevieri în conformitate cu standardele ISO. Codurile asociate limbilor reprezintă două litere mici ale alfabetului, ca de exemplu `"en"` pentru engleză, în timp ce codurile asociate ţărilor reprezintă două litere majuscule, ca de exemplu `"US"` pentru Statele Unite.

Dacă doriţi să vedeţi lista completă a acestor abrevieri, vizitaţi următoarele *site*-uri:

- `http://www.ics.uci.edu/pub/ietf/http/related/iso639.txt`, pentru codurile asociate limbilor;
- `http://www.chemie.fu-berlin.de/diverse/doc/ISO_3166.html`, pentru codurile asociate ţărilor.

Există însă anumite valori predefinite pentru unele ţări (Canada, Franţa, Germania, Italia, Statele Unite etc.) sau pentru unele limbi (engleză, franceză, italiană etc.), identificabile în cadrul clasei `Locale` prin intermediul atributelor statice ale clasei (`CANADA`, `FRANCE` pentru ţări sau, respectiv, `ENGLISH`, `FRENCH`, pentru limbi).

Digitized by Google

În general, operaţiile care necesită internaţionalizarea (formatarea unei date calendaristice, de exemplu), folosesc ca argument o instanţă a clasei `Locale`. Dacă acest lucru nu este specificat, atunci se foloseşte localizarea implicită.

Exemplul 1:

```
//foloseste localizarea implicita in
//operatii de internationalizare
Locale loc = Locale.getDefault();

//afiseaza limba asociata localizarii curente
System.out.println(loc.getDisplayLanguage());

//afiseaza tara asociata localizarii curente
System.out.println(loc.getDisplayCountry());
```

Dacă, de exemplu, localizarea implicită este reprezentată de limba engleză şi Statele Unite, atunci secvenţa de cod anterioară afişează:

```
English
United States
```

Exemplul 2:

```
//creaza o localizare pentru Romania si limba romana
//primul parametru este codul ISO pentru limba romana
//al doilea parametru este codul ISO pentru numele tarii
Locale nl = new Locale("ro", "RO");
```

O localizare poate fi făcută implicită pentru o instanţă a JVM-ului prin metoda `setDefault()`, ca în exemplul următor, în care `nl` este variabila din exemplul precedent):

```
//localizarea romana este setata ca fiind implicita
Locale.setDefault(nl);
```

sau

```
//localizarea germana este setata ca fiind implicita
Locale.setDefault(Locale.GERMAN);
```

Dacă o localizare nu este făcută implicită prin apelul metodei `setDefault`, atunci localizarea implicită este reprezentată de limba în care a fost instalată distribuţia J2SDK.

Timpul poate fi formatat utilizând clasa `DateFormat`. De asemenea, cu ajutorul acestei clase se pot formata şi datele calendaristice sau se pot construi timpul şi datele calendaristice într-o formă independentă de limbă. Clasa oferă numeroase metode statice pentru obţinerea timpului/datelor calendaristice, pe baza localizării implicite sau a uneia specificate explicit, şi pe baza unui

Digitized by Google

număr de stiluri de formatare. Acestea includ opţiunile `FULL`, `LONG`, `MEDIUM`, `SHORT`. În general,

- opţiunea `SHORT` reprezintă data/timpul în forma: 10.12.02 sau 3:30 PM;
- opţiunea `MEDIUM` formatează data/timpul astfel: Dec 12, 2002;
- opţiunea `LONG` formatează data/timpul astfel: December 12, 2002 sau 3:30:55 PM;
- opţiunea `FULL` formatează data/timpul astfel: Thursday, December 12, 2002 AD sau 3:30:55 PM PST.

Pentru a putea formata timpul în localizarea curentă, este necesară obţinerea unei instanţe, folosind metoda `getTimeInstance()` a clasei `DateFormat`. Formatarea propriu-zisă se realizează cu metoda `format()`, ca în exemplul:

```
//afisarea timpului forma scurta in localizarea implicita
DateFormat tf = DateFormat.getTimeInstance(DateFormat.SHORT);
System.out.println(tf.format(new Date()));

//afisarea timpului forma scurta in localizare italiana
tf = DateFormat.getTimeInstance(DateFormat.SHORT,
                                Locale.ITALY);
System.out.println(tf.format(new Date()));
```

Rezultatul execuţiei acestei secvenţe de cod arată astfel:

```
11:39 AM
11.39
```

Prima valoare afişată reprezintă timpul pentru localizarea implicită (engleză în cazul nostru), iar cea de a doua reprezintă timpul în localizare italiană. Diferenţa de formatare dintre cele două valori este cât se poate de clară.

Pentru a formata data calendaristică în localizarea curentă, se folosesc metodele `getDateInstance()` (pentru obţinerea unei instanţe a clasei care permite administrarea datelor calendaristice) şi `format()` (pentru formatarea propriu–zisă), ale clasei `DateFormat`:

```
//afisarea datei forma scurta in localizarea implicita
DateFormat tf = DateFormat.getDateInstance(DateFormat.SHORT);
System.out.println(tf.format(new Date()));

//afisarea datei forma scurta in localizare italiana
tf = DateFormat.getDateInstance(DateFormat.SHORT,
                                Locale.ITALY);
System.out.println(tf.format(new Date()));
```


Digitized by Google

```

//afisarea datei forma scurta in localizare germana
tf = DateFormat.getDateInstance(DateFormat.SHORT,
                               Locale.GERMANY);
System.out.println(tf.format(new Date()));
```

Secvenţa afisează următoarele valori pentru 19 ianuarie 2002, corespunzătoare localizării implicite (engleză, în cazul nostru), localizării italiene (a doua variantă) şi localizării germane (a treia variantă):

```
1/19/02
19/01/02
19.01.02
```

Analog poate fi folosită metoda `parse()`, care, spre deosebire de metoda `format()`, care transformă o dată calendaristică într-un string, realizează operaţia inversă, de creare a unei date calendaristice dintr-un string.

Numerele, procentele şi sumele de bani se formatează utilizând clasa `NumberFormat`, într-un mod asemănător cu cele prezentate anterior.

Aşadar, pentru a formata un număr într-o localizare la alegere se utilizează metodele `getInstance()` şi `format()`:

```
//afisarea numerelor in localizarea implicita
NumberFormat nf = NumberFormat.getInstance();
System.out.println(nf.format(12500.5));

//afisarea numerelor in localizare germana
nf = NumberFormat.getInstance(Locale.GERMANY);
System.out.println(nf.format(12500.5));
```

Rezultatul este:

```
12,500.5
12.500,5
```

Prima valoare afişată este pentru localizarea implicită (engleză în cazul nostru), în timp ce cea de a doua este pentru localizarea germană.

Formatarea procentelor se realizează prin metodele `getPercentInstance()` şi `format()`:

```
//afisarea procentelor in localizarea britanica
NumberFormat nf = NumberFormat.getPercentInstance(Locale.UK);
System.out.println(nf.format(0.3));
```

Rezultatul afişat în urma execuţiei acestei secvenţe este:

```
30%
```


Digitized by Google

Formatarea monetară se realizează prin `getCurrencyInstance()` şi `format()`:

```
//afisarea unei sume de bani in localizarea implicita
NumberFormat nf = NumberFormat.getCurrencyInstance();
System.out.println(nf.format(8599.99));

//afisarea unei sume de bani in localizare italiana
nf = NumberFormat.getCurrencyInstance(Locale.ITALY);
System.out.println(tf.format(8599.99);
```

Rezultatul poate considerat un pic surprinzător:

```
$8,599.99
L. 8.600
```

Surpriza o constituie faptul că pentru lira italiană nu se consideră subdiviziuni, sumele fiind rotunjite pentru a se obţine valori întregi, acest lucru datorându-se faptului că moneda italiană este "slabă" în comparaţie cu celelalte monede. Ca şi în cazul datelor calendaristice, se poate utiliza metoda `parse()` pentru a realiza operaţiile inverse, de transformare a stringurilor în numere.

Probabil că sunt programatori care consideră în acest moment, că formatarea timpului, a datelor calendaristice şi a numerelor sunt doar detalii de afişare a rezultatelor şi nu au implicaţii foarte mari. Utilizatorii aplicaţiilor s-ar putea să nu fie de acord cu această poziţie, în special din cauza faptului că afişarea rezultatelor nu se face în conformitate cu aşteptările lor. De aceea, programatorii trebuie să fie foarte atenţi la acest detaliu pe care, poate, până acum nu l-au tratat cu îndeajuns de multă seriozitate, pentru a realiza o afişare a datelor într-o formă cât mai "prietenoasă".

D.2 Colecţiile de resurse în internaţionalizarea aplicaţiilor

Când programatorii de la Sun au scris codul pentru cele două clase, `DateFormat` şi `NumberFormat`, ei au fost cei care au decis ce localizări să suporte (engleză, franceză, germană etc.) şi au oferit astfel modalitatea de a realiza conversiile de date pentru respectivele localizări. Deoarece variaţiile datelor calendaristice şi ale numerelor nu sunt foarte mari, ei au putut implementa clasele `DateFormat` şi `NumberFormat`, în conformitate cu respectivele localizări, direct în biblioteca de clase (API) pe care Java o oferă.

Din păcate, lucrurile nu au mai fost la fel de simple în cazul mesajelor de tip text. Ar fi fost imposibil pentru programatorii de la Sun să anticipeze toate

Digitized by Google

mesajele de tip text folosite în aplicaţii (existente sau viitoare) şi să ofere traducerile necesare. Din acest motiv, au lăsat această sarcină programatorilor Java care realizează aplicaţii internaţionalizate. Aceasta înseamnă că programatorul Java este cel care decide ce localizări suportă aplicaţia, urmând ca tot el să ofere traducerile mesajelor text pentru fiecare localizare suportată de aplicaţia sa. Cu alte cuvinte, programatorul va crea o colecţie de stringuri pentru o anumită localizare şi le va utiliza în diverse locuri pe parcursul aplicaţiei.

Pentru a realiza acest demers, implementatorii limbajului Java au pus la dispoziţia programatorilor colecţiile de resurse (*engl.* resource bundles), cunoscute altfel şi sub numele de pachete de resurse.

D.2.1 Un exemplu concret

Colecţiile de resurse înglobează toate informaţiile specifice unei anumite localizări. După cum am văzut în paragraful 7.5 (pagina 212), fiecare resursă este asociată cu o cheie, care rămâne neschimbată pentru toate localizările. Valoarea asociată acestei chei se schimbă însă pentru fiecare localizare în parte, reprezentând de fapt traducerea în respectiva limbă a informaţiei deţinute de resursă.

Să considerăm următorul exemplu: dorim să realizăm o aplicaţie care afişează în diverse limbi, noţiunile de calculator, hard-disk, monitor şi tastatură. Deoarece aceste denumiri se vor schimba în cadrul colecţiilor de resurse, fiind înlocuite cu traducerile lor, trebuie să le asociem cu nişte chei care vor rămâne neschimbate. Presupunem că avem următoarele asocieri:

```
Cheie    |   Valoare
----------------------
calc     |   calculator
hdd      |   hard-disk
mon      |   monitor
tast     |   tastatura
```

Asocierile de mai sus formează o colecţie de resurse, cu patru perechi de tipul cheie-valoare. Pentru fiecare limbă în care dorim să traducem aceste noţiuni, vom avea câte o colecţie de resurse în care cheile vor fi aceleaşi cu cele anterioare, iar valorile vor reprezenta traducerile potrivite pentru fiecare noţiune. Vom considera că avem traducerile noţiunilor anterioare în trei limbi: engleză, germană şi franceză. Prin urmare, vom avea alte trei colecţii de resurse, câte una pentru fiecare limbă în care facem traducerea:

- pentru limba engleză

Digitized by Google

```
Cheie    |   Valoare
----------------------
calc     |   computer
hdd      |   hard disk
mon      |   monitor
tast     |   keyboard
```

- pentru limba germană

```
Cheie    |   Valoare
----------------------
calc     |   Computer
hdd      |   Platte
mon      |   Monitor
tast     |   Tastatur
```

- pentru limba franceză

```
Cheie    |   Valoare
----------------------
calc     |   ordinateur
hdd      |   disque dur
mon      |   moniteur
tast     |   clavier
```

După cum se observă, faptul că aceste chei rămân neschimbate ne ajută să asociem corect fiecare noţiune cu traducerea ei. Astfel, noţiunea `calculator` are în primul tabel asociată cheia `calc`, care corespunde în următoarele trei tabele cuvintelor `computer`, `Computer`, `ordinateur`, cu alte cuvinte am obţinut traducerea noţiunii în fiecare dintre cele trei limbi.

Un set de colecţii de resurse formează un grup. Grupul are un nume, ales sugestiv de programator. Analog, fiecare colecţie de resurse din cadrul grupului are un nume, construit în mod unic: numele grupului, urmat de numele limbii în care sunt traduse resursele, de numele ţării şi de *variant* (*variant* reprezintă o subregiune a unei ţări). Ultimele două adăugări de nume sunt opţionale. Numele limbii şi cel al ţării sunt reprezentate prin abrevieri de două litere, în conformitate cu standardele ISO. De exemplu, abrevierea pentru Canada este "CA", iar pentru limba franceză este "fr". În majoritatea cazurilor aceste nume sunt evidente.

Un alt aspect important legat de modul de construire a numelui unei colecţii de resurse este că fiecare nume este separat de următorul prin simbolul `"_"`.

Digitized by Google

Important de reţinut este şi faptul că există o colecţie care are acelaşi nume cu grupul din care face parte. Aceasta este denumită *colecţia implicită*.

Deşi modalitatea de denumire a colecţiilor din cadrul unui grup poate părea dificilă, ea este de fapt destul de simplă. Pentru a verifica această afirmaţie, iată cum arată aceste denumiri în exemplul nostru: am denumit grupul de colecţii `Componente`, ceea ce înseamnă că numele colecţiei implicite este tot `Componente`. Considerăm prima colecţie prezentată (cea în limba română) ca fiind implicită. Cea de a doua colecţie, cea cu traducerile în limba engleză se numeşte, conform regulii de denumire prezentată anterior, `Componente_en`. Analog, celelalte două colecţii se numesc `Componente_de` şi `Componente_fr`.

În continuare vom vedea cum implementează Java colecţiile de resurse sub forma unor subclase ale clasei abstracte `ResourceBundle`:

- `ListResourceBundle`
- `PropertyResourceBundle`

Vom analiza fiecare dintre cele două modalităţi mai detaliat în cele ce urmează.

D.2.2 Implementarea colecţiilor prin `ListResourceBundle`

`ListResourceBundle` conţine resursele în cadrul unor clase Java standard, a căror denumire este dată de numele fiecărei colecţii de resurse, aşa cum a fost el stabilit prin aplicarea regulii de denumire. Aceste clase trebuie să suprascrie metoda `getContents()` şi să ofere un şir care conţine perechile de resurse (cheie, valoare) de tipul `String`. Pentru exemplul nostru, lucrurile stau astfel:

- în primul rând avem colecţia implicită, implementată în modul următor (în fişierul `Componente.java`):

```
import java.util.*;

public class Componente extends ListResourceBundle
{
    static final Object[][] contents = {
                                {"comp", "calculator"},
                                {"hdd", "hard-disk"},
                                {"mon", "monitor"},
                                {"tast", "tastatura"}
                                        };

    public Object[][] getContents()
```


Digitized by Google

```
    {
        return contents;
    }
}
```

- apoi, avem implementarea colecţiei de resurse pentru limba engleză (în fişierul `Componente_en.java`):

```
import java.util.*;

public class Componente_en extends ListResourceBundle
{
    static final Object[][] contents = {
                              {"comp", "computer"},
                              {"hdd", "hard disk"},
                              {"mon", "monitor"},
                              {"tast", "keyboard"}
                                      };

    public Object[][] getContents()
    {
        return contents;
    }
}
```

- mai apoi, implementarea colecţiei de resurse pentru limba germană (în fişierul `Componente_de.java`):

```
import java.util.*;

public class Componente_de extends ListResourceBundle
{
    static final Object[][] contents = {
                              {"comp", "Computer"},
                              {"hdd", "Platte"},
                              {"mon", "Monitor"},
                              {"tast", "Tastatur"}
                                      };

    public Object[][] getContents()
    {
        return contents;
    }
}
```

- în final, implementarea colecţiei de resurse pentru limba franceză (în fişierul `Componente_fr.java`):

Digitized by Google

```
import java.util.*;

public class Componente_fr extends ListResourceBundle
{
    static final Object[][] contents = {
                                {"comp", "ordinateur"},
                                {"hdd", "disque dur"},
                                {"mon", "moniteur"},
                                {"tast", "clavier"}
                                       };

    public Object[][] getContents()
    {
        return contents;
    }
}
```

Odată clasele implementate, ele pot fi utilizate cu succes, conform exemplului din paragraful D.2.4.

Până atunci, iată câteva detalii importante legate de implementarea colecţiilor de resurse prin această metodă. În primul rând, clasele au exact acelaşi nume cu cel stabilit la definirea colecţiilor: `Componente`, `Componente_en`, `Componente_de`, `Componente_fr`. În al doilea rând, informaţia de asociere a cheilor cu valorile corespunzatoare este conţinută în interiorul claselor. Fiecare pereche cheie/valoare reprezintă un element al şirului `contents`. Din cauza faptului că datele sunt stocate în interiorul unor clase, apare un dezavantaj destul de important: clasele trebuie recompilate pentru orice modificare a valorilor. Este motivul pentru care această variantă este mai puţin utilizată de programatori.

O alternativă a acestei clase este clasa `PropertyResourceBundle`, care reţine resursele în fişiere de proprietăţi.

D.2.3 Implementarea colecţiilor prin `PropertyResourceBundle`

Clasa `PropertyResourceBundle` foloseşte un tip special de fişiere pentru a stoca informaţiile de mapare a cheilor şi a valorilor. Aceste fişiere sunt de fapt simple fişiere text, cu extensia `.properties`. Numele fiecărui fişier de acest tip este dat tot de regula de denumire a colecţiilor de resurse. Aşadar, în exemplul nostru sunt patru fişiere de proprietăţi cu următoarele denumiri:

- `Componente.properties`
- `Componente_en.properties`

Digitized by Google

- `Componente_de.properties`
- `Componente_fr.properties`

Formatul special al fişierelor de proprietăţi apare datorită faptului că perechile cheie/valoare sunt stocate pe câte o linie separată. Fiecare linie este formată din cheie, urmată de semnul `"="`, iar în final este adăugată valoarea asociată cheii.

Pentru exemplul nostru, cele patru fişiere `.properties` au următorul conţinut:

- fişierul `Componente.properties`:

```
comp=calculator
hdd=hard-disk
mon=monitor
tast=tastatura
```

- fişierul `Componente_en.properties`:

```
comp=computer
hdd=hard disk
mon=monitor
tast=keyboard
```

- fişierul `Componente_de.properties`:

```
comp=Computer
hdd=Platte
mon=Monitor
tast=Tastatur
```

- fişierul `Componente_fr.properties`:

```
comp=ordinateur
hdd=disque dur
mon=moniteur
tast=clavier
```

În acest moment, fişierele de proprietăţi sunt gata pentru a fi utilizate.

Digitized by Google

D.2.4 Utilizarea colecţiilor de resurse

Indiferent că aţi optat pentru prima metodă de implementare a colecţiilor de resurse (utilizând clasa `ListResourceBundle`), sau pentru cea de a doua (prin intermediul clasei `PropertyResourceBundle`), utilizarea colecţiilor de resurse se realizează în acelaşi mod. Aşadar, dacă doriţi să treceţi de la o metodă la cealaltă, nu trebuie să modificaţi nimic în modul de utilizare a colecţiilor de resurse.

Această secţiune va arăta cât de elegant se face traducerea noţiunilor alese dintr–o limbă în alta. Pentru aceasta, vom considera următorul program de test (fişierul `UtilRB.java`):

```
import java.util.*;

public class UtilRB
{
    public static void main(String[] args)
    {
        try
        {
            Locale.setDefault(new Locale("ro", "RO"));
            ResourceBundle rb = ResourceBundle.getBundle(
                    "Componente", Locale.getDefault());

            String s = rb.getString("comp");
            System.out.println(s);

            s = rb.getString("tast");
            System.out.println(s);
        }
        catch(Exception e)
        {
            System.out.println("EXCEPTION: " +
                                        e.getMessage());
        }
    }
}
```

Prezentat pe scurt, programul anterior setează localizarea implicită ca fiind cea română, iar apoi utilizează colecţia de resurse ataşată localizării implicite (cu alte cuvinte, colecţia de resurse implicită, `Componente`), pentru a descoperi valorile ataşate celor două chei `"comp"` şi `"tast"`, valori pe care le afişează. Rezultatul execuţiei acestui program este:

```
calculator
tastatura
```

Analog se pot afişa valorile celorlalte chei.

Digitized by Google

Pentru a vedea adaptabilitatea foarte bună a colecţiilor de resurse la modificarea limbilor folosite, realizaţi în programul sursă anterior modificările necesare pentru ca programul să arate astfel:

```
import java.util.*;

public class UtilRB
{
    public static void main(String[] args)
    {
        try
        {
            Locale.setDefault(new Locale("ro", "RO"));

            ResourceBundle rb = ResourceBundle.getBundle(
                                "Componente", Locale.ENGLISH);

            String s = rb.getString("comp");
            System.out.println(s);

            s = rb.getString("tast");
            System.out.println(s);
        }
        catch(Exception e)
        {
            System.out.println("EXCEPTION: " +
                                     e.getMessage());
        }
    }
}
```

Modificarea constă în faptul că s–a schimbat localizarea, adică limba curentă în care se fac afişările de date este cea engleză. Dat fiind faptul că localizarea curentă este cea engleză, la rularea programului, Java va prelua valorile cheilor `"comp"` şi `"tast"` din colecţia de resurse asociată limbii engleze, adică din colecţia `Componente_en`. În urma execuţiei, vor fi afişate rezultatele:

```
computer
keyboard
```

Analog se poate obţine traducerea noţiunilor în celelalte două limbi pe care le–am ales: germană şi franceză. Pentru a realiza acest demers, trebuie doar să înlocuiţi `Locale.ENGLISH` în programul anterior cu `Locale.GERMAN` şi mai apoi cu `Locale.FRENCH`.

Un lucru important care trebuie amintit este legat de modul în care Java identifică pachetul de resurse pentru o anumită localizare. Colecţia este căutată în cadrul grupului din care face parte, conform paşilor algoritmului următor,

Digitized by Google

căutarea încheindu–se în momentul în care este găsită colecţia căutată sau s–a ajuns la colecţia implicită.

1. bundleName_localeLanguage_localeCountry_localeVariant
2. bundleName_localeLanguage_localeCountry
3. bundleName_localeLanguage
4. bundleName_defaultLanguage_defaultCountry_defaultVariant
5. bundleName_defaultLanguage_defaultCountry
6. bundleName_defaultLanguage
7. bundleName

`bundleName` reprezintă numele grupului, iar `localeLanguage`, `localeCountry` şi `localeVariant` se referă la limba, ţara şi regiunea localizării specificate, în timp ce `defaultLanguage`, `defaultCountry` şi `defaultVariant` se referă la localizarea implicită.

Pentru a înţelege mai bine acest algoritm de căutare a colecţiilor de resurse, vom arăta cum se realizează căutarea în exemplul nostru. Mai întâi să identificăm numele din algoritm: `bundleName` este în exemplul nostru `Componente`, `localeLanguage` este fie `en` (engleza), fie `de` (germana), fie `fr` (franceza), în timp ce `localeCountry` şi `localeVariant` lipsesc. Pe de altă parte, `defaultLanguage` este `ro` (româna), în timp ce `defaultCountry` şi `defaultVariant` lipsesc de asemenea.

Dacă, de exemplu, dorim să căutăm colecţia de resurse pentru limba germană, trebuie să căutăm colecţia cu numele `Componente_de`, cu alte cuvinte `bundleName` ia valoarea `Componente`, iar `localeLanguage` ia valoarea `de`. Pentru că nu avem în această situaţie `localeCountry` sau `localeVariant`, algoritmul va fi executat până la pasul 3, când se opreşte pentru că a găsit colecţia cu numele `Componente_de`.

Într–un mod asemănător, dacă dorim să căutăm colecţia de resurse pentru limba franceză vorbită în Canada (`new Locale("fr", "CA")`), Java va căuta colecţia cu numele `Componente_fr_CA`. Executând paşii algoritmului anterior, ne vom opri la pasul 3, pentru că nu există o colecţie numită `Componente_fr_CA` (şi din acest motiv nu ne putem opri la pasul 2), dar există una numită `Componente_fr`. Aşadar, în această situaţie vom avea traducerile din limba franceză standard, şi nu traducerile pentru dialectul de limbă franceză vorbit în Canada.

Digitized by Google

Forţând un pic nota, dacă dorim să căutăm colecţia de resurse pentru limba italiană (`Locale.ITALIAN`), Java caută colecţia de resurse cu numele `Componente_it`, dar cum nu avem nici o componentă cu nume asemănător, căutarea se va încheia la pasul 7, la colecţia implicită `Componente`, traducerile fiind preluate din această colecţie. Prin urmare, dacă nu avem o colecţie asociată unei localizări specificate, traducerile vor fi oferite din colecţia implicită.

Folosind tehnicile prezentate în primele două secţiuni ale acestei anexe, se pot construi aplicaţii simple care afişează utilizatorului date în orice localizare. Datele pot fi valori ale timpului, date calendaristice, numere sau simple mesaje statice, cum ar fi cele prezentate anterior. Dacă dorim să folosim mesaje dinamice (de genul: `"x fişiere au fost şterse"`), lucrurile se modifică destul de mult. Secţiunea următoare arată cum trebuie tratate mesajele de acest tip.

D.3 Internaţionalizarea mesajelor dinamice

Din perspectiva programatorului care rescrie aplicaţii într–o altă limbă, cuvintele sunt cea mai simplă parte ce trebuie implementată. De exemplu, când se traduc cuvintele din engleză în germană, `"Yes"` devine `"Ja"`, `"No"` devine `"Nein"` şi aşa mai departe. Chiar şi frazele simple, de tipul `"Do you want to save the modifications?"`, pot fi înlocuite în întregime cu traducerile potrivite. Dacă traducerile s–ar rezuma doar la exemple simple, de tipul celor menţionate anterior, atunci nu ar fi probleme. Dar ce facem dacă suntem nevoiţi să traducem un mesaj de tipul: `"You copied 2 files"`? Spre deosebire de mesajele anterioare, acesta este un mesaj dinamic, conţine informaţii care se modifică în funcţie de alte operaţii.

O metodă de rezolvare ar putea fi împărţirea mesajului în mai multe părţi, în aşa fel încât partea dinamică a mesajului să fie despărţită de cea statică. Cu alte cuvinte să împărţim mesajul în trei părţi: `"You copied"`, `"2"` şi `"files"`. În acest fel, partea dinamică a mesajului (`"2"`) este separată de cea statică. Apoi, ar urma ca cele două părţi statice să fie preluate din colecţii de resurse, unde ar fi traduse separat. Practic, codul ar fi asemănător cu:

```
ResourceBundle rb = ResourceBundle.getBundle("Strings");
String str = rb.getString("first") + nFiles +
             rb.getString("second");
```

unde `"first"` şi `"second"` ar fi cele cheile asociate celor două părţi statice ale mesajului, iar `nFiles` ar fi numărul de fişiere ce va fi copiat.

Digitized by Google

Din păcate, abordarea precedentă nu este universal valabilă, deoarece este posibil ca în alte limbi ordinea cuvintelor să nu mai fie aceeaşi. Deşi noi am considerat că elementele care compun textul au o anumită ordine, există limbi în care această ordine nu poate fi respectată. De exemplu, în limba germană verbele se pun la finalul propoziţiei.

Din fericire, există o soluţie pentru această problemă. Trebuie să creăm un şablon corect din punct de vedere sintactic pentru fiecare localizare, iar apoi la executarea aplicaţiei şablonul este combinat cu partea dinamică a mesajului (altfel spus, cu parametrii). Abilitatea de a combina şablonul cu parametrii este oferit în Java prin intermediul clasei `MessageFormat`.

Primul pas care trebuie realizat este crearea câte unui şablon pentru fiecare localizare suportată de aplicaţie. Şablonul presupune înlocuirea părţilor dinamice ale mesajului, cu parametrii, ca în exemplul:

```
"You copied 2 files" --> "You copied {0} files"
```

În cazul nostru, singura parte dinamică a mesajului este `"2"` şi a fost înlocuită cu parametrul `"{0}"`. Dacă ar fi existat mai multe părţi dinamice în cadrul mesajului, parametrii ar fi fost `{1}`, `{2}` etc.

Considerăm şablonul anterior ca fiind asociat localizării implicite (în cazul nostru, localizarea engleză). De asemenea, considerăm şi şablonul pentru localizarea română:

```
"Ati copiat {0} fisiere"
```

La executarea aplicaţiei, parametrul `{0}` este înlocuit în cadrul şablonului, cu numărul de fişiere copiate, aşa cum a fost el calculat în cadrul aplicaţiei.

Cele două şabloane, câte unul pentru fiecare dintre cele două localizări, vor fi preluate din colecţii de resurse şi folosite pentru a crea mesajele dinamice. Considerăm că numele grupului de colecţii este `"Messages"`, iar cheia asociată şablonului este `"msg"`.

Colecţia de resurse implicită este următoarea (`Messages.properties`):

```
msg=You copied {0} files
```

Pe de altă parte, colecţia de resurse pentru limba română arată astfel (fişierul `Messages_ro.properties`):

```
msg=Ati copiat {0} fisiere
```

Pentru localizarea implicită, secvenţa de cod arată astfel:

Digitized by Google

```
ResourceBundle rb = ResourceBundle.getBundle("Messages");

MessageFormat mf = new MessageFormat(rb.getString("msg"));
Object[] args = {new Integer(2)};

System.out.println(mf.format(args);
```

La execuţia codului va fi preluat din colecţia implicită şablonul `"You copied {0} files"`. Prin apelului metodei `format()`, şablonul este combinat cu valorile aflate în variabila `args`. Practic, parametrii şablonului sunt înlocuiţi cu valorile de pe poziţiile corespunzătoare din şirul `args`. În cazul nostru, parametrul `{0}` este înlocuit cu `args[0]`, adică cu elementul `2`, obţinându–se mesajul formatat `"You copied 2 files"`.

Pentru obţinerea traducerii acestui mesaj în limba română, modificaţi prima linie a secvenţei de cod anterioare, astfel încât să fie de forma:

```
ResourceBundle rb = ResourceBundle.getBundle("Messages",
                                    new Locale("ro", "RO"));
```

Analog celor prezentate anterior, se va obţine un mesaj de forma `"Ati copiat 2 fisiere"`.

Exemplele anterioare folosesc cele mai simple forme de parametri în cadrul unui şablon. Java oferă posibilitatea de a realiza formatări mult mai complexe.

Un parametru este format de fapt din trei câmpuri separate prin simbolul `","`:

- Primul câmp reprezintă un număr, care indică poziţia din cadrul şirului de obiecte, a obiectului cu care va fi înlocuit parametrul la formatare. De exemplu, parametrul 0 indică faptul că va fi înlocuit cu elementul de pe poziţia 0 din şir, parametrul 1 cu elementul de pe poziţia 1 etc.;

- Al doilea câmp este opţional şi se referă la tipul obiectelor formatate. Poate avea una din următoarele valori: `time`, `date`, `number`, `choice`;

- Al treilea câmp este de asemenea opţional şi se referă la stilul de formatare a obiectelor. Dacă formatul stabilit la punctul anterior este `time` sau `date`, atunci stilul poate avea valorile: `short`, `medium`, `long`, `full` sau un stil de dată definit de utilizator. Dacă formatul este `number`, atunci stilul poate avea valorile: `currency`, `percentage`, `integer`, sau un stil de număr definit de utilizator.

Iată câteva exemple de parametri:

```
{0}              - formateaza elementul de pe pozitia
```


Digitized by Google

```
                     0 din sir
{0,time}           - formateaza elementul de pe pozitia
                     0 din sir ca reprezentand timpul
{0,date,short}     - formateaza elementul de pe pozitia
                     0 din sir ca fiind o data
                     calendaristica in format scurt
(0,number,percent) - formateaza elementul de pe poz.
                       0 din sir ca fiind un procent
```

Formatul `choice` este mai special şi îl vom descrie în continuare. Opţiunea `choice` (din lista de opţiuni pentru câmpul al doilea al unui parametru) permite ataşarea de mesaje unui anumit domeniu de valori. Opţiunea este utilă pentru că poate modifica mesajele în funcţie de anumite valori ale parametrilor. Dacă în exemplul nostru, valoarea parametrului ar fi fost 1 sau 0, mesajul ar fi arătat astfel:

```
"You copied 1 files"
```

sau

```
"You copied 0 files"
```

Prima propoziţie nu este corectă din punct de vedere gramatical (`files` este la plural) şi Java ne ajută să evităm o astfel de situaţie, prin folosirea opţiunii `choice`. Opţiunea ne oferă posibilitatea de a modifica mesajul în funcţie de valoarea parametrului, astfel încât să avem următoarele mesaje:

- dacă parametrul are valoarea 0, atunci mesajul este `"You copied no files."`;
- dacă parametrul are valoarea 1, atunci mesajul este `"You copied 1 file."`;
- dacă parametrul are valoarea > 1, atunci mesajul este `"You copied` *N* `files."`, unde *N* este un număr.

Sintaxa opţiunii `choice` este descrisă în continuare. Opţiunea este specificată ca un set de alternative, separate prin simbolul `"|"`. O alternativă constă dintr-o limitare numerică şi un mesaj. Alternativele sunt ordonate după limitarea numerică. Limitarea reprezintă un număr `double`, care poate fi specificat în două moduri:

Digitized by Google

- *N#*, ce înseamnă că pentru toate valorile mai mari decât *N*, inclusiv, mesajul este valabil;
- *N*<, ce înseamnă că pentru toate valorile mai mari decât *N*, exclusiv, mesajul este valabil.

Folosind opţiunea `choice`, şablonul mesajului nostru arată astfel:

```
"You copied {0,choice,0#no files|1#1 file|
                      1<{0} files}"
```

Deşi pare destul de complicată, la o privire mai atentă se poate obseva că utilizarea `choice` nu este chiar atât de dificilă. Şablonul ne dezvăluie că formatarea se aplică parametrului 0, adică elementului de pe poziţia 0 din şirul de elemente, că este o formatare de tipul alegere (*engl.* choice) cu trei alternative, separate prin două simboluri `"|"`. Prima alternativă indică faptul că mesajul `"no files"` va fi afişat pentru valori ale parametrului ≥ 0 şi < 1 (conform limitării numerice a celei de a doua alternative), adică pentru valoarea 0. Asemănător, alternativa a doua indică faptul că mesajul `"1 file"` este afişat pentru o valoare a parametrului între ≥ 1 (limitarea numerică a alternativei a doua) şi ≤ 1 (limitarea numerică a alternativei a treia), adică pentru valoarea 1 a parametrului. Alternativa a treia indică faptul că pentru valori ale parametrului > 1, se afişează mesajul "*N* `files`", unde *N* este valoarea parametrului.

După cum se poate observa, limitările numerice ale alternativelor sunt ordonate crescător (0, 1, 1).

Cu prezentarea formatării mesajelor dinamice se încheie şi ultima etapă care trebuie parcursă pentru a putea crea aplicaţii internaţionalizate la standarde profesionale. Ajuns la acest nivel, programatorul Java trebuie să conştientizeze faptul că internaţionalizarea aplicaţiilor nu mai reprezintă o irosire a timpului, ci chiar o necesitate.

Digitized by Google

E. Resurse Java

Important este să nu încetezi
niciodată să îţi pui întrebări.

Albert Einstein

E.1 *Site*-ul Sun Microsystems

Pagina principală a tuturor resurselor Java este `http://java.sun.com`, realizată de creatorii limbajului Java, unde veţi găsi informaţii complete despre acest limbaj. *Site*-ul reprezintă o colecţie impresionantă de documentaţii, tutoriale, cărţi despre Java. Sugestia noastră este ca în situaţia în care aveţi nevoie de informaţii suplimentare despre limbajul Java, să începeţi căutarea dumneavoastră aici.

Iată câteva adrese utile în cadrul *site*-ului:

- `http://java.sun.com/docs` - documentaţii Java;
- `http://java.sun.com/products/jdk/faq.html` - răspunsuri la întrebări adresate frecvent;
- `http://java.sun.com/docs/books` - cărţi despre limbajul Java;
- `http://java.sun.com/docs/books/tutorial` - tutorialul complet al limbajului Java;
- `http://java.sun.com/j2se` - pagina principală a platformei `J2SE`;
- `http://java.sun.com/products` - pagina de informaţii despre principalele produse ale Sun Microsystems.

Digitized by Google

E.2 Tutoriale despre limbajul Java disponibile pe Internet

Tutoriale Java pot fi găsite la următoarele adrese:

- `http://www.artima.com`
- `http://www.zdnet.com`
- `http://directory.google.com/Top/Computers/Programming/Languages/Java/FAQs,_Help,_and_Tutorials/Tutorials`
- `http://developer.java.sun.com/developer/onlineTraining/Programming`
- `http://www.javacoffeebreak.com/tutorials`
- `http://webreference.com/programming/java/tutorials.html`
- `http://freewarejava.com`
- `http://www.1netcentral.com/online-tutorials.html`
- `http://www.programmingtutorials.com/tutorials.asp?id=Java`
- `http://javaboutique.internet.com`

E.3 Cărţi despre limbajul Java disponibile pe Internet

Marea majoritate a *site*-urilor care prezintă cărţi despre limbajul Java oferă doar o mică descriere a acestora, pentru a crea o impresie de ansamblu celor care doresc să cumpere cărţile respective. Există însă şi cărţi complete disponibile gratuit. Pentru ambele categorii vom prezenta câteva adrese utile:

- `http://java.sun.com/docs/books`
- `http://www.informit.com`

Digitized by Google

- `http://gshulkin.hypermart.net/books/FreeJavaBook.htm`
- `http://freewarejava.com/books/index.shtml`
- `http://www.better-homepage.com/java/java-books.html`
- `http://www.mindview.net`, unde poate fi găsită gratuit una dintre cele mai bune cărţi despre Java, "Thinking in Java", scrisă de Bruce Eckel.

E.4 Reviste online de specialitate

Apariţia Internetului a însemnat şi o prezentare a revistelor într-un nou format, cel electronic, mult mai accesibil decât cel standard (cel puţin din punct de vedere al distanţei, deoarece oricine poate citi o revistă care apare în SUA chiar din momentul apariţiei ei). Comunitatea IT se bucură de prezenţa a numeroase reviste online de prestigiu, care oferă atât informaţii generale din domeniul IT cât şi despre limbajul Java în special:

- `http://www.javaworld.com` - liderul revistelor online care tratează limbajul Java
- `http://www.pcmag.com`
- `http://www.slashdot.org` - revistă online care prezintă informaţii generale din domeniul IT şi nu numai
- `http://www.enews.com`
- `http://www.infoworld.com`

E.5 Liste de discuţii despre limbajul Java

Pentru a comunica online (prin email) cu alţi programatori, pe tema limbajului Java, vă recomandăm înscrierea la lista de discuţii *Advanced Java*, găzduită de Universitatea Berkeley:
`http://lists.xcf.berkeley.edu/mailman/listinfo/advanced-java`

Digitized by Google

E.6 Alte resurse Java

În final, vă prezentăm o listă de *site*-uri unde puteţi găsi diverse alte informaţii despre limbajul Java:

- `http://www.ibiblio.org/javafaq`
- `http://www.gamelan.com`
- `http://www.apl.jhu.edu/~hall/java`
- `http://www.web-hosting.com/javalinks.html`

Digitized by Google

Bibliografie

[Andonie] R. Andonie, I. Gârbacea - Algoritmi şi Calculabilitate, Computer Press Agora, 1995

[Balanescu] T.Bălănescu - Metoda Backtracking, Gazeta de informatică, nr 2/1993, pag. 9-18

[Boian] F.M. Boian - De la aritmetică la calculatoare, Presa Universitară Clujeană, 1996, ISBN 973-97535-5-8

[Cormen] T.H. Cormen, C.E. Leiserson, R.R. Rivest - Introducere în Algoritmi, Computer Press Agora, 2000, ISBN 973-97534-3-4

[Danciu] D.Danciu, S.Dumitrescu - Algoritmică şi Programare, Curs şi Probleme de Seminar, Reprografia Univ. Transilvania, 2002

[Eckel] B. Eckel - Thinking in Java, http://www.mindview.net/Books /TIJ/

[Horrowitz] E. Horrowitz, S. Sahni - Fundamentals of Computer Algorithms, Computer Science Press, ISBN 3-540-12035-1

[Norton] P. Norton, W. Stanek - Peter Norton's Guide to Java Programming, Sams.net Publishing, 1996, ISBN 1-57521-088-6

[Roman] Ed Roman - Mastering Enterprise JavaBeans, Wiley Computer Publishing, 1999, ISBN 0-471-33229-1

[Sorin] T.Sorin - Tehnici de programare, manual pentru clasa a X-a, L&S Infomat, 1996

[Tomescu] I. Tomescu - Data Structures, Bucharest University Press, Bucharest, 1997

Digitized by Google

[Vanderburg] G. Vanderburg - Tricks of the Java Programming Gurus, Sams.net Publishing, 1996, ISBN 1-57521-102-5

[Weiss] M.A. Weiss - Data Structures and Problem Solving Using Java, Addison-Wesley, 1998, ISBN 0-201-54991-3

[Sundsted] T. Sundsted - Internationalize dynamic messages, JavaWorld article (www.javaworld.com)

[JavaTutorial] Sun Microsystems - The Java Tutorial (java.sun.com/docs/books/ tutorial)

[Pal] G. Pal - Exceptions in Java. Nothing exceptional about them, JavaWorld article (www.javaworld.com)

[JavaDocs] Sun Microsystems - Java docs (java.sun.com/docs)

Digitized by Google

Index

Digitized by Google

Digitized by Google

Digitized by Google

Digitized by Google

Digitized by Google

Digitized by Google

Digitized by Google

Digitized by Google